The Fastest Verification

ZeBu™
zRun Emulation Interface
- User Manual -

Document revision – d –

December 2010

Version 6_3_0

Copyright Notice
Proprietary Information

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

Figures

About This Manual

Overview

This manual describes the ZeBu **zRun** Emulation Interface. It describes the graphical interface and how to use it, and how to use Tcl commands to control emulation runs without using the graphical interface.

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
d	6_3_0	Dec 10	New sections: • Special case for control of emulation without GUI, (2.2.6.2). • *Flp on Trig* feature (§3.5.3). • SystemVerilog Assertions (§3.9). • Deselecting signals from a Tcl script (§4.4.8). • Using the *Flp on Trig* feature (4.7.7). • Controlling dynamic forces (§4.10). • Controlling DPI function calls (§4.11). Updated sections: Several sections have been updated, in particular: • **Global Control** panel, added new **SVA** button (§3). • **Run Control** panel, added clock count for ZeBu-Server (§3.4). • **Run Control** panel → **Clock Group** frame, updated max. cycle count (§3.4.2). • **Flexible Local Probes** panel → **Waveform Dump** frame (§3.7.2), **nWave (streamed)** mode not available in V6_3_0. • Opening **zRun** session with `-do` option and Tcl script (§4.1.1). • Getting signal information, updated signal types list (§4.4.4). • Pseudo-randomizing signals/memories, updated description of `ZEBU_randomizeSignals`, `ZEBU_randomize`, and `ZEBU_randomizeMemories` (§4.12.2). New commands: • `ZEBU_Signal_deselectAllSignals` (§4.4.8). • `ZEBU_Signal_isForceable` (§4.10). • `ZEBU_isHDP` (§4.14.14). Updated commands: • `ZEBU_dumpFile` (§4.8.1), updated syntax. • `ZEBU_selectSignalsToRandomize` and `ZEBU_selectMemoriesToRandomize`, added description for `<insert>` argument (§4.12.1). • `ZEBU_Flp_setSamplingClock` (§4.13.3), updated syntax. • `ZEBU_Flp_setOutputDir` (§4.13.4), updated syntax.
c	4_3_3	Feb 09	General update: Flexible Probes and ZEMI-3 features.
b	4_3_2	Jan 09	General update to match V4_3_2 features.
a	1_3_0	June 05	First edition.

Related Documentation

- The ***ZeBu Compilation Manual*** describes the complete steps of the compilation process.
- The ***ZeBu HDL Co-simulation Manual*** describes the use of the HDL co-simulation driver for the ZeBu platform.
- The ***ZeBu C++ Co-simulation Manual*** describes the use of the C++ co-simulation driver for ZeBu platform.
- The ***ZeBu SystemC Co-simulation Manual*** describes the use of the SystemC co-simulation driver for ZeBu platform.
- Additional ZeBu documentation is listed in Chapter 5.

Typographic Conventions Used in This Manual

ZeBu tools are shown in bold, mono-space "Courier New" font, for example **zRun**.

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) 2002-2008                    #
# 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

zRun is a Tcl-based Graphical User Interface (GUI) which controls the ZeBu system emulation runtime. A Tcl command set, listed in Chapter 4, can be used in addition to the **zRun** GUI.

zRun is generally used with the following verification modes:

- Co-simulation with transaction-level C/C++
- Co-simulation with signal-level C/C++/HDL
- Emulation with a synthesizable testbench
- Emulation with in-situ connection

zRun provides the following functions:

- Clock control
- Dynamic probes monitoring (including register writing)
- Flexible probes monitoring
- Memory edition
- Logic analysis
- Waveform file generation
- Signal tracing
- Tcl command line interpreter
- SystemVerilog Assertions (SVA)

2 Launching zRun

This chapter first lists the possible options that you can use in the **zRun** command line. It then describes the typical **zRun** commands that you are likely to use when running an emulation.

2.1 zRun options

Option (case insensitive)	Description
`-attach [PID]`	Starts **zRun** while a co-simulation is running.
`-debugDriverClk`	Allows you to control the `driverClk`. It is mainly used to debug transactor emulations or to generate a dump file in asynchronous mode (see §3.4.2.2).
`-design designFeatures`	Specifies a `designFeatures` file for an embedded testbench.
`-do <tclScript>`	Executes a user-defined Tcl script on startup.
`-functionList`	Prints the list of Tcl commands used by **zRun**.
`-gui <guiScript>`	Executes the GUI Tcl script.
`-help`	Prints the **zRun** help file.
`-logFile <fileName>`	Defines the log filename.
`-noclockRefresh`	Runs without refreshing the displayed clock counters.
`-noGui`	Runs **zRun** without the GUI (in this case, you can also use the `-do <tclScript>` option to execute a user-defined Tcl script).
`-noLog`	Does not generate any log file.
`-synchroClose`	Synchronizes testbench closure with **zRun** closure.
`-testBench <testbench>`	Runs an optional co-simulation testbench (only for C/C++ co-simulation or transaction-based verification).
`-zebu.work <zebu.work>`	Optional path to compilation directory for an embedded testbench. Default is `./zebu.work`. This option is not taken into account in case you also use the `-testBench` option.

Note: the `DISPLAY` environment variable must be set correctly before you start the **zRun** program, especially if you use a remote terminal.

2.2 zRun command line

2.2.1 Using zRun with a C/C++ testbench

If you use **zRun** with a C/C++ testbench, you must specify a testbench file after the **zRun** command and the `-testBench` option.

```
$> zRun -testBench myTestBench
```

Note: zRun controls the emulation, including the start of the clocks.

2.2.2 Starting zRun during co-simulation

You can start **zRun** while a co-simulation is running using the following command:

```
$> zRun -attach [PID]
```

Where `PID` is the Linux process identification of **zServer** in the current session. The PID is not necessary if you have a single emulation running on the ZeBu system and no process waiting in the ZeBu connection queue.

2.2.3 Using zRun without a testbench

When you use **zRun** without a C/C++ testbench, use the `designFeatures` file to configure the ZeBu system. This file contains definitions for different items such as clock configuration, name/path of memory initialization file(s), etc.

Use the `-design` option in the **zRun** command line to specify the `designFeatures` file. You must indicate the working directory of the design using the `-zebu.work` option (most of the time `./zebu.work`).

```
$> zRun -design designFeatures -zebu.work ./zebu.work
```

2.2.4 Controlling the emulation with a Tcl script before zRun GUI is launched

The `-do <tclsript>` option allows you to specify a Tcl script which will be launched at the opening of **zRun**. This is useful when an initialization phase is to be executed before the opening of the **zRun** GUI.

The Tcl script can contain any Tcl command, as well as the specific **zRun** commands listed in Chapter 4. You can set up probes, initialize memories, and automate the entire run using Tcl commands. Example with `designFeatures` and aTcl script:

```
$> zRun -design designFeatures -zebu.work ./zebu.work -do init.tcl
```

2.2.5 Controlling the emulation using your own GUI Tcl script

Use the `-gui` option to execute your own GUI Tcl script instead of the default script available at `$ZEBU_ROOT/etc/tcl/zGui.tcl`.

```
$> zRun -design designFeatures -zebu.work ./zebu.work -gui ./my_gui.tcl
```

2.2.6 Controlling the emulation without the graphical interface

2.2.6.1 Normal usage

Use the `-nogui` option to execute a user-defined Tcl script with **zRun** in batch mode. Note that **zRun** controls the emulation, including the start of the clocks.

```
$> zRun -design designFeatures -zebu.work ./zebu.work -do init.tcl -nogui
```

2.2.6.2 Special case

The `-nogui`, `-do`, and `-testbench` options can all be used in the same command:

```
zRun -nogui -do script.tcl -testbench "./tb"
```

However, special attention should be paid to the following conditions. The testbench and the **zRun** script must be synchronized to make sure that the closing of the testbench does not invalidate the last commands in the script, namely `Dump_off` commands, etc.

- USE the following command :

  ```
  zRun -synchroClose
  ```

  After the testbench is closed with `zebu->close()`, an explicit closing of **zRun** is expected (via `-nogui` in the script or via the **Close** button in the GUI).

- DO NOT USE the following in the script:

  ```
  while { [ZEBU_getStatus]=="open" &&
    [ZEBU_Clock_getStatus clk]=="running" } { after 100 }
  ```

  `[ZEBU_getStatus]=="open"` is not useful (script closes the connection).
  USE the following instead:

  ```
  while { [ZEBU_Clock_getStatus clk]=="running" &&
    [ZEBU_synchroCloseStatus] == "false" } { after 100 }
  ```

  Exemple:

  ```
  ZEBU_Dump_file monitor.vcd clk
  ZEBU_Dump_on
  ZEBU_Clock_enableForever clk
  while { [ZEBU_Clock_getStatus clk]=="running" &&
     [ZEBU_synchroCloseStatus] == "false" } { after 1000 }
  ZEBU_Clock_disable clk
  ZEBU_Dump_off
  ZEBU_close
  ZEBU_exit
  ```

3 zRun Graphical Interface

3.1 Default graphical interface

The default GUI is displayed when **zRun** is launched with no emulation in progress. It displays the **Global Control** panel with access to a limited number of features, as shown in the following figure.

The following functions are available:

Open	Opens the connection to the ZeBu system by loading the runtime database of the selected design and initializing it for the run.
Restore	Restores the hardware state of your design from a previously saved file.
Dyn Change	Launches **zSelectProbes** to add dynamic probes to the design, as described in Section 3.3.4.
Exit	Closes the GUI and exits **zRun**.

Figure 1: Default zRun GUI

All functions of the **Global Control** panel are described in Section 3.3.

3.2 Complete graphical interface

When a session is open and connection has been established with the ZeBu system, the **zRun** window is updated to take into account all elements of the design.

Figure 2 shows a typical interface after a connection is established. This example includes all features of the **zRun** GUI. The panels which are visible when you open a session with **zRun** depend on your design, and may be slightly different from the ones illustrated in Figure 2.

There are 4 panels and a command line:

- **Global Control** panel
- **Run Control** panel
- **LA Control** panel (Logic Analyzer control)
- **Memory Control** panel
- **Tcl Command Line**

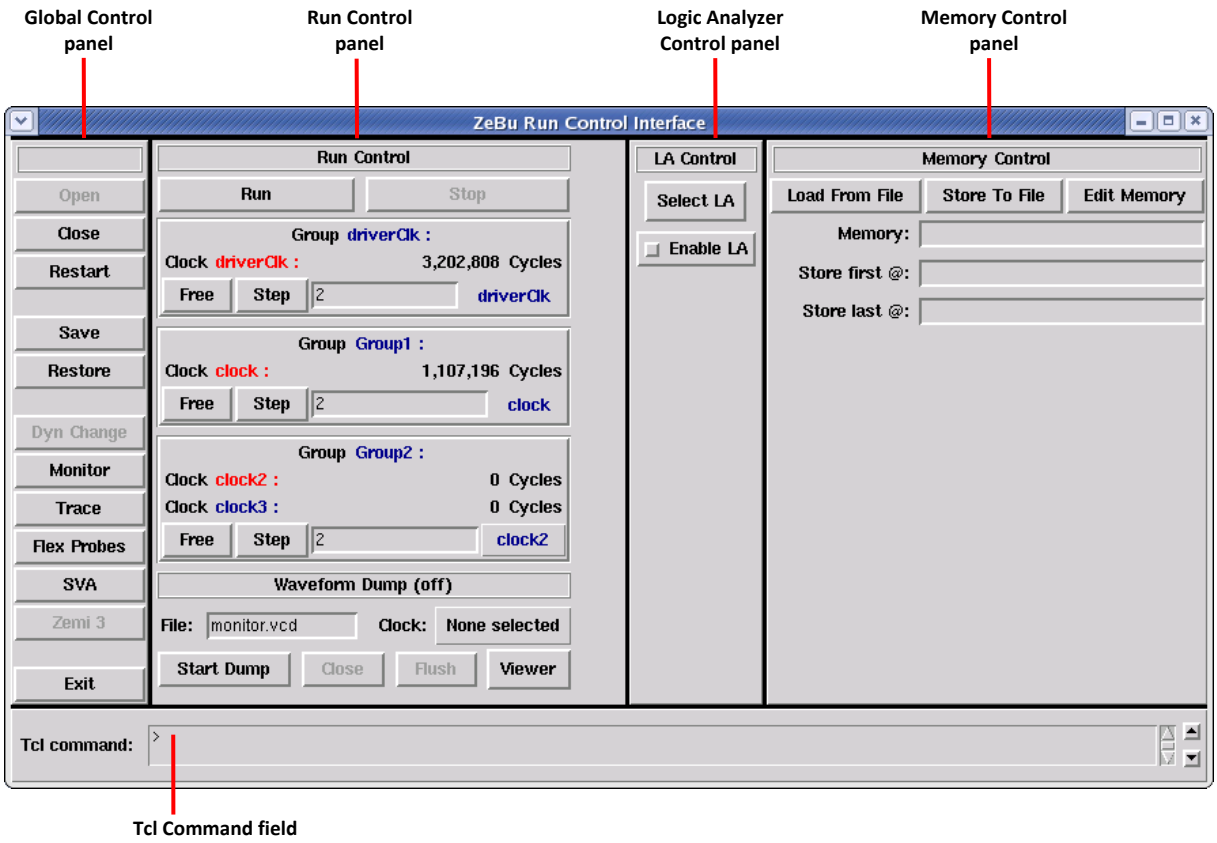


Figure 2: zRun GUI main window after connection

3.3 Global Control panel

3.3.1 Description

The global commands are the following:

Open Opens the connection to the ZeBu system by loading the runtime database of the selected design and initializing it for the run.

Close Closes the connection to the ZeBu system and keeps the GUI open.

Restart Closes and opens the connection to the ZeBu system and re-initializes the design. Use it for a new emulation session. See **WARNING** below.

Save Saves the hardware state of design in a file (§3.3.2).

Restore Restores the hardware state of design from a file previously saved (§3.3.2).

Monitor Opens a signal browser and a monitor to select the signals that you will read/write at runtime (§3.3.5).

Trace Opens a panel to control Trace memory (§3.3.3).

Flex Probes Opens a panel to control Flexible Probes (§3.7).

SVA Opens a panel to control SystemVerilog Assertions (§3.9).

Zemi 3 Opens a panel to control ZEMI-3 transactors(§3.8).

Exit Closes the GUI and exits **zRun**.

Figure 3: Global commands

WARNING (for the **Restart** button)**:** if another user was previously in queue when you use this function, his job will start and you will be queued following his job!

3.3.2 Save & Restore feature

Save & Restore is a feature you can use to capture the state of the design at runtime. The state of the design can be restored in the ZeBu system during the same emulation, or during a later emulation.

Figure 4: Access to Restore in zRun default GUI

Figure 5:Access to Save & Restore in zRun complete GUI

You can restore a state of the design either before you open a session or during a run.

When you click on **Save**, a folder is created which contains the state of the DUT FPGAs, memories, and design clocks. When you click on **Restore**, the session is automatically closed (if necessary) and re-opened with restoration of a previously saved state of the design.

3.3.3 Trace Without Trig feature

It is possible to trace the signals specified in the `SRAM_TRACE` (which is instantiated in the DVE file) independently of the triggers. In this mode you can store the last cycles before the emulation stopped or get a state of design without decreasing the emulation frequency.

Notes:

- This function is not supported if your ZeBu system is a Software Development Platform (SDP) and it is only available if you have instantiated `SRAM_TRACE` in the DVE file (Section 4.4.8).
- By default, 256 MBytes are reserved in each FPGA module for SRAM trace purposes. It is possible to modify the reserved memory capacity during the initialization phase of the ZeBu-Server system. Any modification impacts the entire ZeBu-Server system. See details in ***ZeBu-Server Installation Manual***.

Click **Trace** in the **Global Control** panel, or disable **Logic Analyzer** if **Trace on Trig** is selected. To enable/disable trace during emulation, use the **Trace** button. The **Trace Memory Usage** panel indicates a percentage of the trace memory to be used.

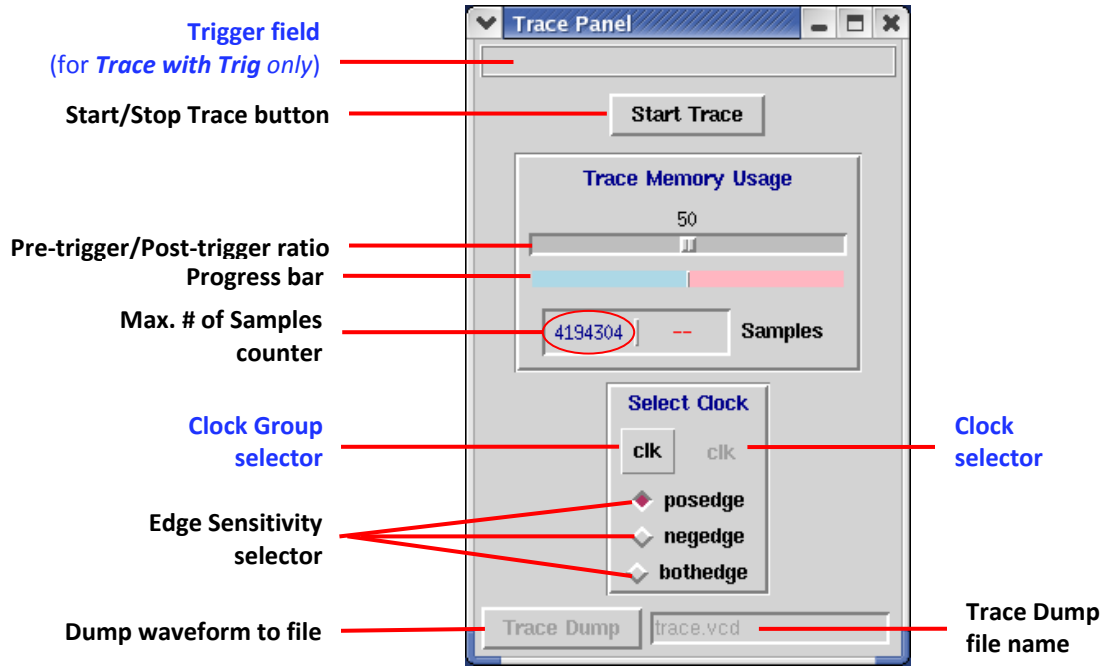


Figure 6: ***Trace Panel*** **for** ***Trace without Trig***

The **Start/Stop Trace** button has two states:

- **Start Trace**
- **Trace Enabled**: when trace is running, click this button to stop it.

Trace Memory Usage frame:

- **Pre-trigger/Post-trigger** ratio: use slider to vary percentage of trace memory.
- **Pre-trigger Progress** bar (light blue): progress is shown in dark blue.
- **Post-trigger Progress** bar (light pink): progress is shown in dark pink.
- **Max. # of Samples** counter**:** indicates the maximum number of samples which can be captured and stored, according to the **Pre-trigger/Post-trigger** ratio.

Select Clock frame:

- **Clock Group** selector: click to select a sampling clock group.
- **Clock** selector**:** click to select a sampling clock from the current clock group. This button is grayed out if the group does not contain more than one clock.
- **Edge Sensitivity** selector**:** set one of 3 edge sensitivities for the selected clock.

The **Trace Panel** for *Trace without Trig* is similar to the **Trace Panel** for **Trace on Trig**. The only difference is the **Trigger** field which is empty for *Trace without Trig*.

Using the Trace feature is very simple. For instance, you can:

- Click the **Start Trace** button even while emulation is running.
- Run a certain number of cycles (at least one cycle with the selected clock).
- Click the **Stop Trace** button even while emulation is running.
- Store the results in the selected waveform file (`VCD`, `BIN` or `FSDB` format) even while emulation is running.
- Open the waveform file with an appropriate viewer such as GTKWave.

Notes:

- `FSDB` is a format used by Novas™, and requires Debussy® or nWave™ for viewing. This format is more compact than `VCD`. To write `FSDB` files, add the path of the `libnffw.so` dynamic library to the `LD_LIBRARY_PATH` variable: `<novas root>/share/FsdbWriter/<architecture>`.
- **Hard-Wired Mode:** In case you selected a sampling clock at compilation time (through a DVE file), then the SRAM trace is in hard-wired mode. You cannot select this clock at runtime. See *Special Case* in Section 4.4.8 for details.

3.3.4 Selecting dynamic probes

The **Dyn Change** button can only be used when the connection with ZeBu is closed. Use this button to add dynamic probes to your design. This way, the next time a connection is established with ZeBu, you will be able to access internal signals without exiting **`zRun`**.

To add dynamic probes to your design when the **`zRun`** GUI is active:

1. If the connection to the ZeBu system is open, click **Close** to disconnect.
2. Click **Dyn Change** to launch **`zSelectProbes`**.
3. Once in **`zSelectProbes`**, select the `zebu.work` path.

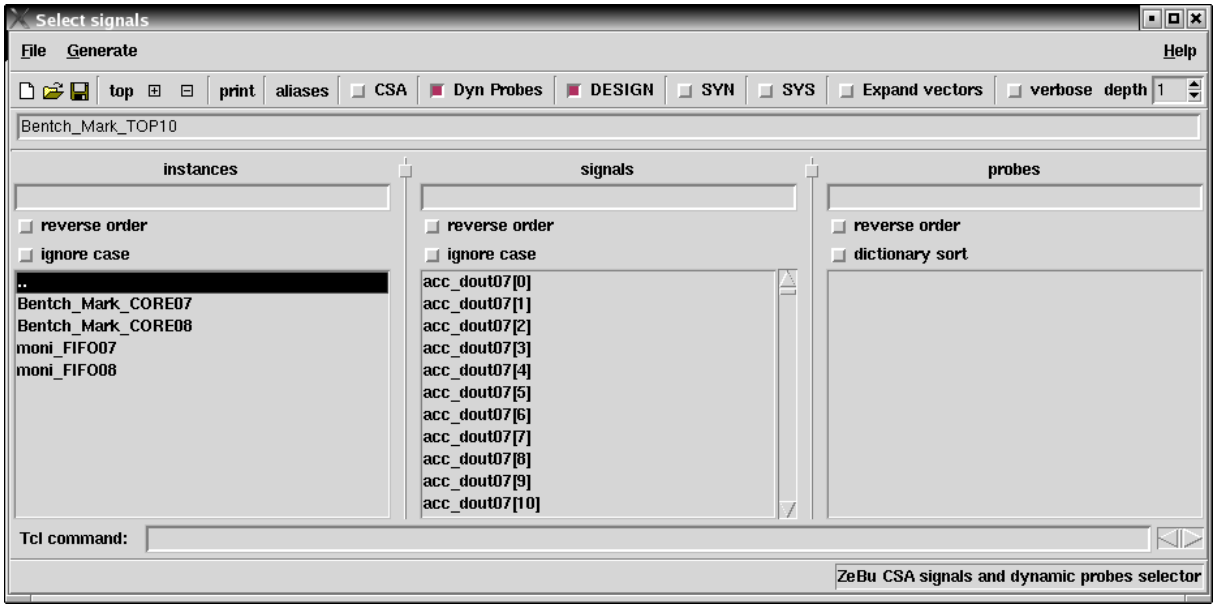


Figure 7: `zSelectProbes` graphical interface

4. Add your dynamic probes using the browser function of **`zSelectProbes`**:
   a. Double-click on any instance name in the design hierarchy pane (left) to display (in the same pane) the instances declared within the current instance. The available ports/signals are displayed in the middle pane.
      - To display Xilinx primitives, select **View > Xilinx Primitives**.
      - To view aliases you have created, select **View > EDIF Renamed**.
   b. Click on the signal/port you want to add then click **Add [+].** To add all signals/ports, click the instance name (left pane) then **Add [+]**.
   c. Click **Save** to save your selection of probes.
   d. In the **Generate** menu, select the wrapper type to generate for the probes. Go back to the **Generate** menu and select **Generate**. The appropriate probe file(s) are now available, according to your test environment.
   e. Select **File > Exit** to close the **`zSelectProbes`** graphical interface.
5. In **`zRun`**, click **Open** to re-open the connection to the ZeBu system.

The dynamic probes can now be accessed/modified using the **Monitor** browser.

3.3.5 Monitor function

Use this function to select the signals that you want to read/write during emulation. This can be done when the clocks are stopped, as well as during the run without affecting the emulation frequency.

3.3.5.1 Description

Click the **Monitor** button in the **Global Control** panel to open 2 panels at once:

- **Signal Hierarchy Browser**
- **Signal Monitor**

Note: The emulation frequency is slowed down when the **Update** button is clicked while the clocks are running.

Dynamic probes can be added using **`zSelectProbes`**, as described in Section 3.3.4. With the **Signal Hierarchy** browser, you may want to visualize dynamic probes only or visualize all probes (in this case, you must check the **Show All Probes** box).

- Signals that you can read/force are displayed in black.
- Signals that are read-only are displayed in red or green.

Note: You can also add probes on-the-fly without using **`zSelectProbes`**.

3.3.5.2 Signal Hierarchy Browser

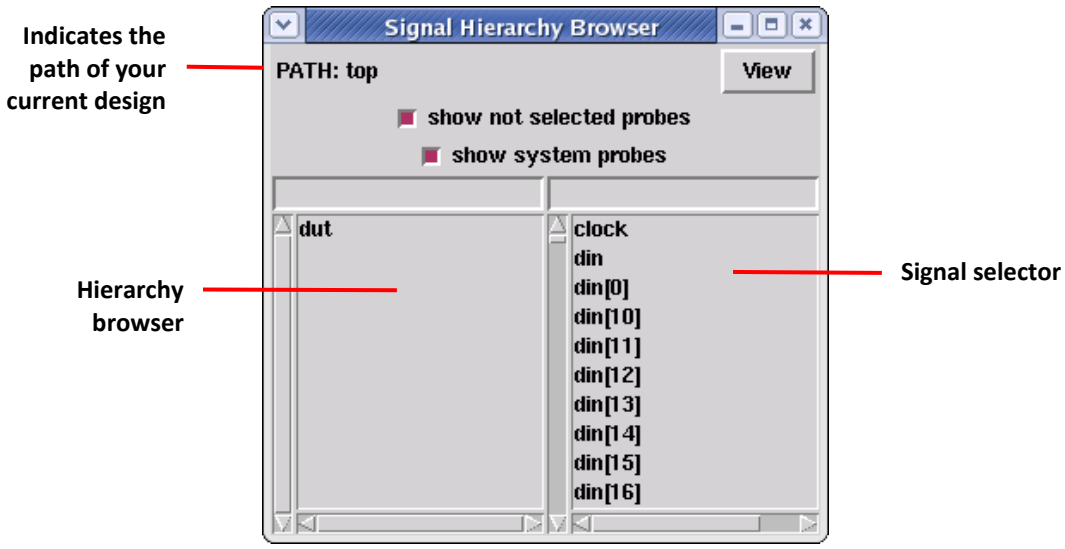


Figure 8: *Signal Hierarchy Browser*

PATH:	The top line in the **Signal Hierarchy Browser i**ndicates the hierarchical level of your current design with an extra top level that encompasses all your hierarchy.
View button	Click this button to automatically add all signals from the current path hierarchy and sub-hierarchy levels to the **Signal Monitor**.
Show not selected probes checkbox	Check this box to show all signals available as dynamic probes and not previously selected with **`zSelectProbes`**.
Show system probes checkbox	Check this box to show also the dynamic probes inserted during synthesis.
Hierarchy browser	Displays the hierarchy of the signals in the design.
Signal selector	Displays the signals available in the current level of the hierarchy. Choose the signals you want to add to the **Signal Monitor** (described in section 3.3.5.3).

To add signals to the **Signal Monitor**:

To view a single signal	Double-click on a signal name or select a signal name then click the **View** button.
To view several signals	While holding the SHIFT or CTRL key down, click on the signal names in the **Signals** panel then click the **View** button.

Note: If a hierarchy path is selected in the left pane and **View** is clicked, all signals from the current hierarchy level and from its sub-hierarchy levels are added to the **Signal Monitor**.

3.3.5.3 Signal Monitor

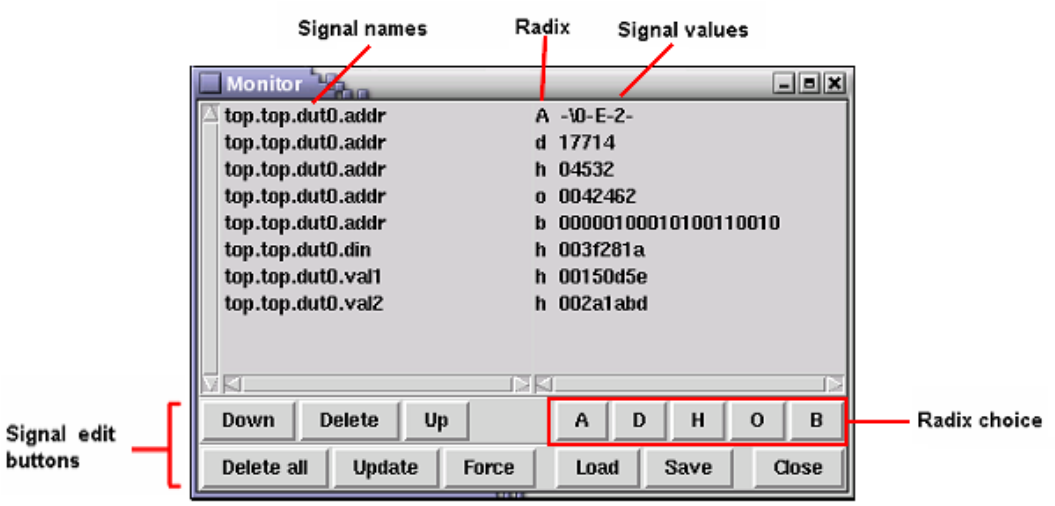


Figure 9: *Signal Monitor*

The signals are displayed with their name, radix and value. A single signal can be displayed several times with different radices (for example `top.top.dut0.addr` in Figure 9).

Top-level interface signals and combinational dynamic probes are displayed in green. The following functions are available in the **Signal Monitor**:

Down	Moves a selected signal down in the list of signals.
Delete	Removes a selected signal from the list.
Up	Moves a selected signal up in the list of signals.
A,D,H,O,B	Sets the radix of the displayed value for the currently selected signal: A = ASCII, D = Decimal, H = Hexadecimal, O = Octal, B = Binary.
Delete all	Deletes all signals from the list.
Update	Updates the display. Normally, the values shown here are for a run that has ended. However, if you have selected **Free** (Run forever mode), use **Update** to verify that signals are changing values. This is a capture mode, which does not affect the emulation clocks frequency.
Force	Flushes the signals with the values that you enter using the **Write Signals panel** described in Section 3.3.5.4. Note that when forcing ASCII signals, they will be processed in hexadecimal format.
Load	Opens a **File Open** browser to load into the **Signal Monitor** a list of signals that was previously saved.
Save	Opens a **File Save** browser to save the displayed list of signals.
Close	Closes the **Signal Monitor** and the **Signal Hierarchy Browser**.

3.3.5.4 Forcing signals to given values

You can force one or more signals to required values. Ensure that you have the correct radix selected for the signal(s) that you want to force.

Note: When forcing ASCII signals, they will be processed in Hexadecimal format. In the example below, clicking the **Force** button for ASCII value `@@` will display `4040` in the **Write Signals** panel which is the hexadecimal equivalent of `@@`.

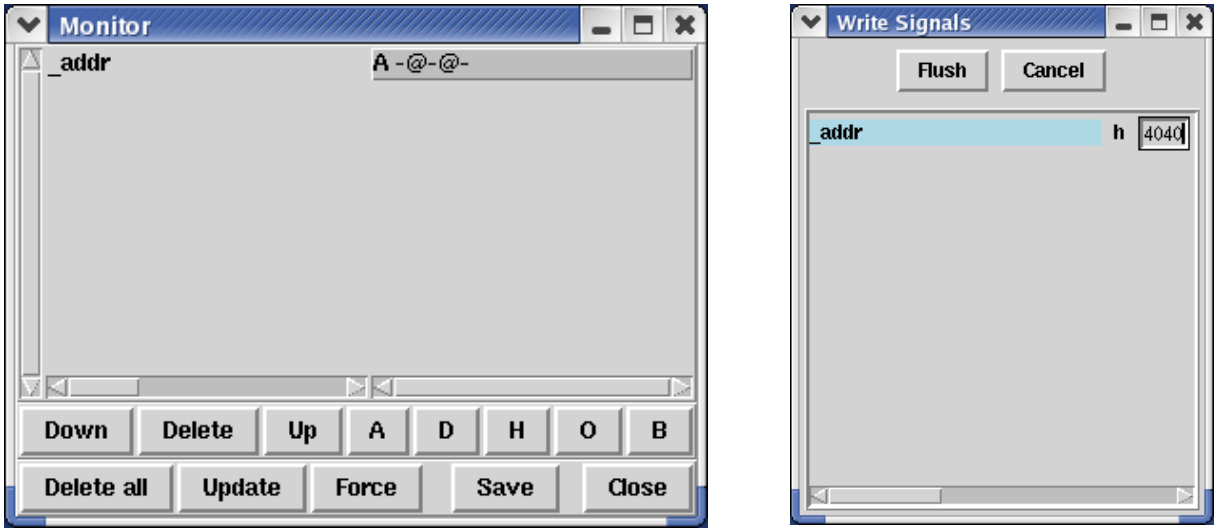


Figure 10: Forcing an ASCII signal to a different value

Signals displayed in red (top-level interface signals) or green (combinational dynamic probes) cannot be forced. An error message is displayed when you click the **Force** button for such signals.

To force a single signal	Double-click on a signal name in the **Signals** panel to open the **Write Signals** panel, or select a signal name then click the **Force** button.
To force several signals	While holding the SHIFT or CTRL key down, click on the signal names in the **Signals** panel then click the **Force** button.

1. Enter the value(s) for the signal(s) in each signal field.
2. Click **Flush**. The **Write Signals** panel closes and the **Signal Monitor** is updated with new signal values.

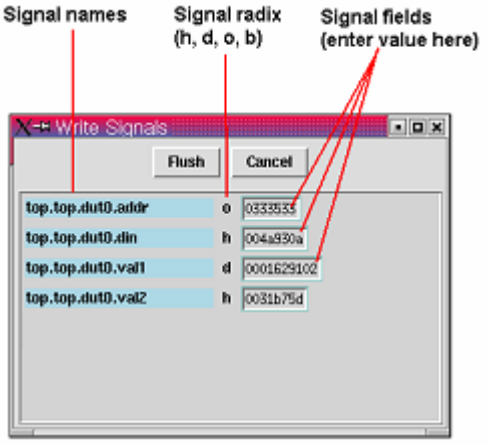


Figure 11: *Write Signals* panel

3.4 Run Control panel

The **Run Control** panel contains at least three frames: **Run/Stop**, **Clock Groups** (one frame for each clock group), and **Waveform Dump**.

Figure 12 shows an example of the **Run Control** panel with 3 clock groups with 4 clocks: `driverClk` has its own group, `Group1` (`clock`) and `Group2` (`clock2` and `clock3`). There can be up to 9 clocks in V4_3_x (8 + `driverClk`) and up to 17 clocks in V5_1_0/V6_3_0 (16 + `driverClk`), organized into groups of one or more clocks.

Note: `driverClk` is included in the clock count if **zRun** was launched with the `-debugDriverClock` option.

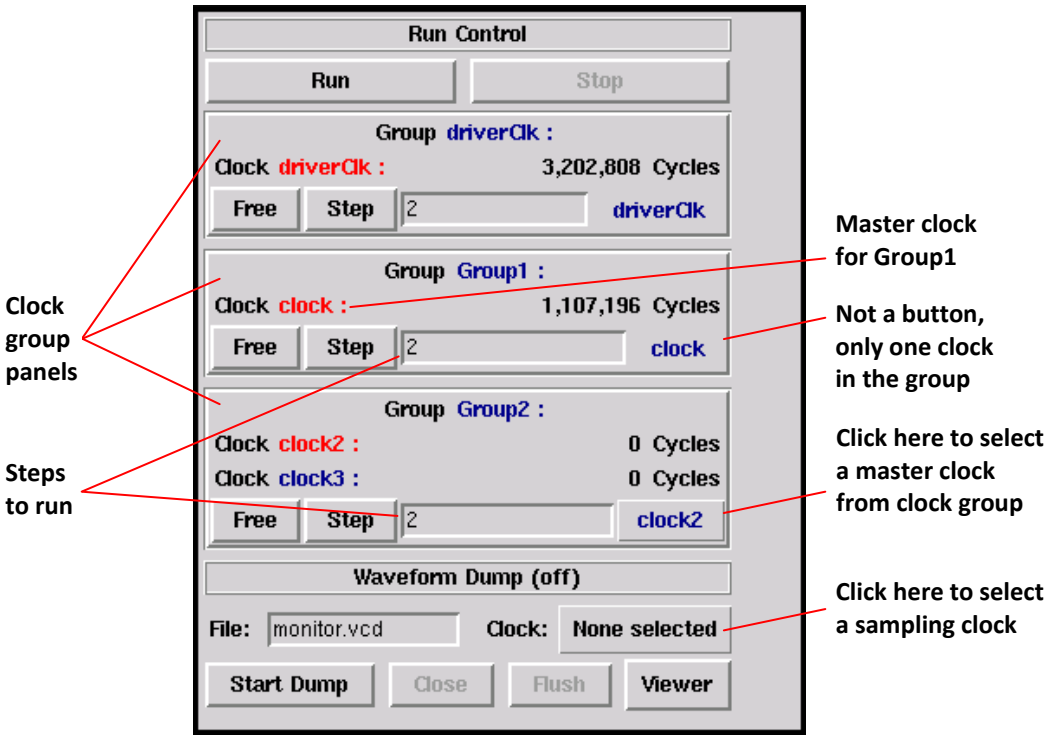


Figure 12: ***Run Control*** **panel**

3.4.1 Run/Stop Control buttons

Figure 13: ***Run/Stop Control*** **buttons**

Run Starts the run. All selected clocks are activated simultaneously. This button is grayed out when the run is in progress. Enter a number of clock cycles in the **Steps to Run** field for each clock (See Figure 14); if this value is `0` for a clock, this clock doesn't run. The run ends automatically when the last selected clock stops.

Stop Interrupts the current run. All clock groups stop simultaneously *except* for clocks that are in free-running mode. This button is only active when a run is in progress.

3.4.2 Clock Group Frames

3.4.2.1 Clock control

In most cases, you will only use one clock group, but you can also have several groups. There is a **Clock Group** frame for each clock group. Each **Clock Group** frame (Figure 14) displays the following information:

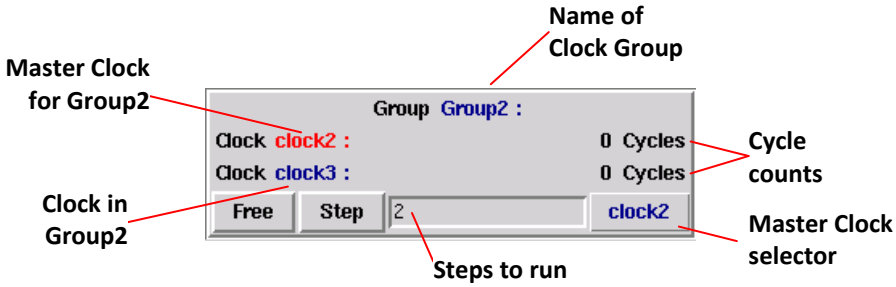


Figure 14: *Clock Group* frame

Note: The master clock of the group is displayed in red in the **Clock Group** frame. A clock is called master clock when it starts/stops all clocks of the same group with one **Start/Stop** command. In a clock group, clocks can have different frequencies and waveforms and only the master clock is associated to the **Steps to Run** field.

Group name Name of the clock group (`Group2` in this example). This is the name entered in the `designFeatures` file.

Clock names List of clocks defined for the group: `clock2` and `clock3; clock2` has been set as the master clock for the group. It is displayed in red.

Cycle count Number of clock cycles executed (since the start of the emulation) by the master clock of this group. During the run this number is refreshed automatically except when the `-noclockRefresh` option is used. The maximum number of clock cycles ranges into the hundreds of billions of cycles.

Free The currently selected clock group runs without any interruption. Click the **Free** button to start or stop a free run.

Step Advances the current master clock step-by-step (one cycle at a time). Each time you click the button the clock advances one cycle. This function is independent of the value of the **Steps to Run** field.

Steps to Run Enter the number of clock steps that the current master clock will execute when **Run** is clicked. 0 means that the clock will not run. When **Run** is clicked, the master clock will execute the specified number of cycles. All other clocks in the same group will execute a number of cycles determined by their respective clock virtual frequencies (specified in the `designFeatures` file).

Master Clock Selector Selects the master clock for the group. This button is identified by `Clock2` in Figure 12.

3.4.2.2 Additional clock control

If the `-debugDriverClk` option is used in the **zRun** command line, a special group called `driverClk` is displayed: this is the system clock control. When this option is declared, all clocks can be started simultaneously in free-running mode. It is mainly used to debug transactor emulations or generate a dump file in asynchronous mode. To start the clocks simultaneously, follow these steps:

1. Click on **Free** in the clock groups frames you want except `driverClk`.
2. Click on **Free** in the `driverClk` clock group frame.

All the clocks which belong to the selected clock groups will start simultaneously. When `driverClk` stops, all the clocks stop simultaneously (except external clocks).

3.4.3 Waveform Dump Frame

In the **Waveform Dump** frame, you can control the options and enable waveform dumping of the selected probes when all clocks are stopped.

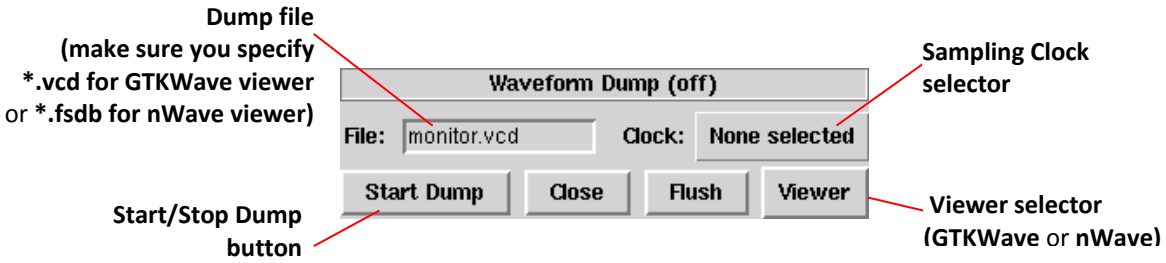


Figure 15: ***Waveform Dump*** **frame**

Dump File	You can dump the traces into the selected waveform dump file in `VCD`, `VPD`, `BIN` or `FSDB` format. The default format is `VCD` and the default filename is `monitor.vcd`. You can edit the filename when it is not grayed out (right-clicking on the filename opens a file browser).
Start/Stop Dump	Starts/Stops waveform dumping in the specified dump file.
Flush	During the waveform dump, flushes the waveform into the specified dump file without closing it. This is useful when you use the nWave streamed mode so that the viewer updates its content accordingly.
Viewer	Click this button to select a waveform viewer via a drop-down menu. Make sure you specify a `.vcd` file extension for your dump file before you select the GTKWave viewer, or a `.fsdb` file extension before you select the nWave viewer.

The first step is to specify the sampling clock in the **Sampling Clock selector**. There are 2 different sampling modes:

- **Asynchronous mode**: The sampling clock is `driverClk` which is the fastest clock in the design. This sampling mode will only be displayed if **zRun** was launched with the `-debugDriverClk` option.
- **Synchronous mode**: You can select a design clock as the signal sampling clock.

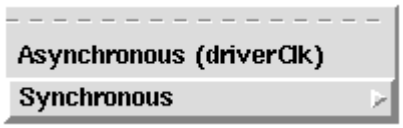


Figure 16: Selecting the sampling mode

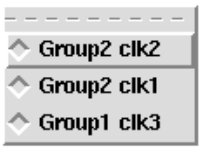


Figure 17: Selecting a synchronous clock

3.4.4 Example

In this figure, `Clock2` of `Group2` has been selected for **Waveform Dump**, thus `Clock2` becomes the master clock (if it is not already the case).

Waveform Dump is not yet used, so you can change the master clock if necessary, and you can enter a new name for the waveform file.

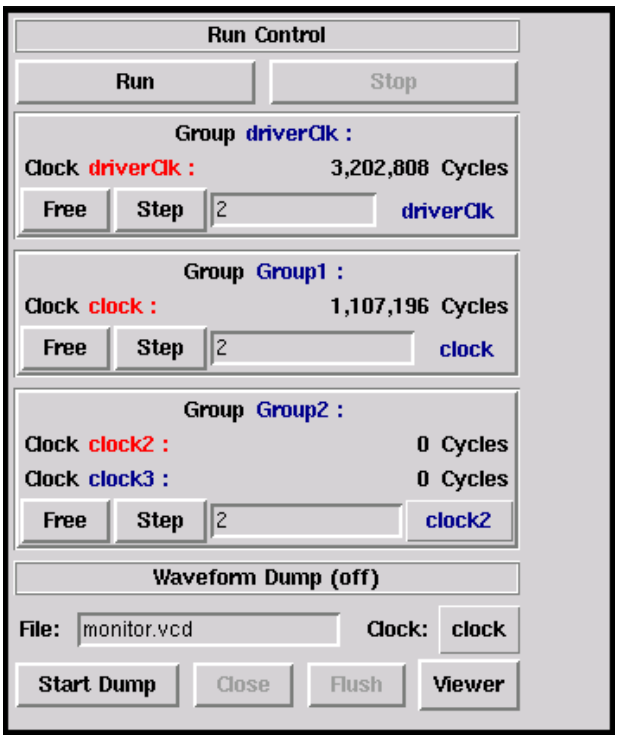


Figure 18: *Run Control* (*Waveform Dump* not used)

When you click **Start Dump** for the first time, a waveform dump file is created and the dump begins the next time the **Free** or **Step** button is pressed for the selected sampling clock.

Once the waveform dump has started, it is not possible to change the dump file, the selected sample clock, or any of the master clocks unless you restart the emulation.

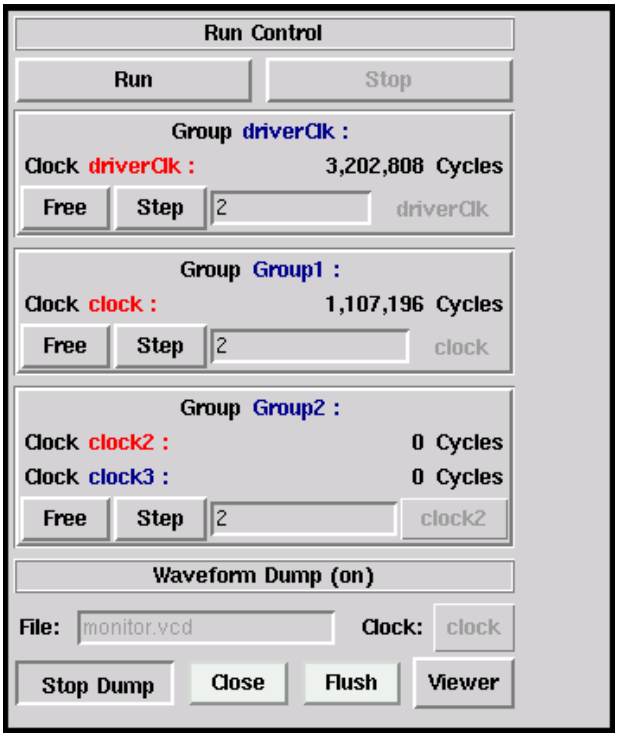


Figure 19: ***Run Control*** **(*Waveform Dump* active)**

In this figure, all clocks are in **Free** running mode.

Do not enable or disable waveform dumping during the emulation if clocks are in free-running mode. If you do so, the VCD output will not be relevant (inaccurate waveforms).

Figure 20: ***Run Control*** **(*Free-Running* mode)**

When the emulation is not running, you can enable or disable waveform dumping.

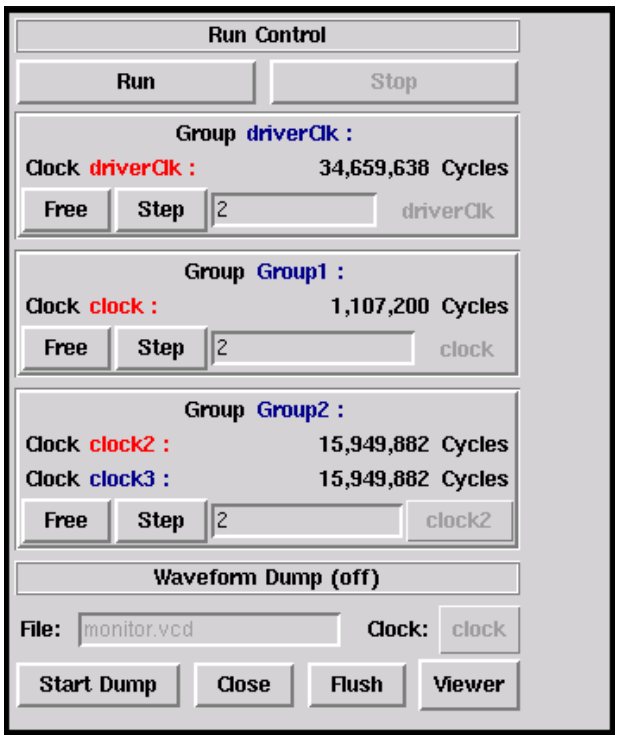


Figure 21: *Run Control* (Emulation not running)

3.5 LA Control panel

The **LA Control** panel (Logic Analyzer control) is only available if you have declared static or dynamic triggers in the DVE file (see ***ZeBu-Server Compilation Manual***).

To select the type of logic analyzer to use with your trigger, click **Select LA**:

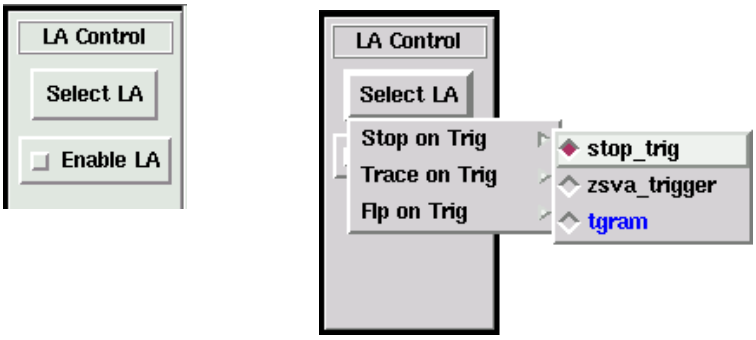


Figure 22: *LA Control* → *Select LA* menu

When selecting a trigger for the **Logic Analyzer** feature, note that static trigger names are in displayed in black, and dynamic trigger names in blue.

There are three types of **Logic Analyzer** functions:

Stop On Trig (SOT)	Stops the emulation on either a static or dynamic trigger.
Trace On Trig (TOT)	Controls SRAM_TRACE with static or dynamic triggers. This function is only available if you have instantiated SRAM_TRACE in the DVE file. **This function is not available if your ZeBu system is a Software Development Platform (SDP)**.
Flp on Trig (FOT)	Activates the dump feature from flexible probes on any trigger event declared in the DVE file.

Note: If a Logic Analyzer was enabled when you use the **Restart** button in the **Global Control** panel, it will remain enabled after the restart.

3.5.1 *Stop On Trig* feature (SOT)

3.5.1.1 Usage

Use the **Stop on Trig** menu item to stop emulation when the selected trigger is fired.

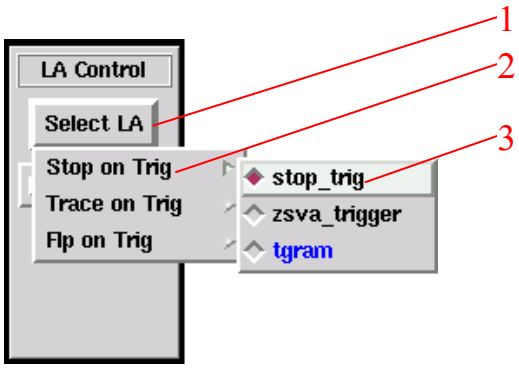


Figure 23: ***LA Control → Stop On Trig*** **menu item**

1. Click **Select LA**.
2. From the pop-up menu, click on **Stop on Trig**.
3. Select a trigger name.

In case of a dynamic trigger, specify an expression you want to associate to the trigger according to the list of signals attached to it in the DVE file.

3.5.1.2 Example for Dynamic Trigger

In case of a dynamic trigger, specify an expression you want to associate to the trigger according to the list of signals attached to it in the DVE file. In the example below, the dynamic trigger is called `dut_din_31to0`.

Figure 24: Dynamic trigger expression for Logic Analyzer

The **LA Control** panel is updated with the **Select LA** button replaced by the **Stop on dut_din_31to0** button, and a **Clock Selector** where you can:

- Select a sampling clock to be associated with the trigger.
- Set the clock edge sensitivity.

To select a sampling clock:

1. Select clock group: click the **Clock Group** button (for ex. `Group2`).
2. Select sampling clock: click the **Clock** button (for ex. `clk1`).
3. Select the edge sensitivity (`posedge` or `negedge`)

When the trigger is fired, all clocks are stopped. You can restart a run without disabling the Logic Analyzer but it will still be valid for the next trigger.

Figure 25: ***LA Control → Stop On Trig*** **panel**

3.5.2 *Trace On Trig* feature (TOT)

3.5.2.1 Usage

Use the **Trace on Trig** feature to control the way the signals sampled with `SRAM_TRACE` are stored in the trace memory. The signals are traced before and after the selected trigger fires. You can also specify the following:

- Number of samples before the specified trigger fires (pre-trigger memory).
- Number of samples after the specified trigger fires (post-trigger memory).

Before trigger occurs	Signals are dumped into pre-trigger memory in the same way as a circular buffer: when the data exceeds pre-trigger memory size, old data is overwritten. Pre-trigger memory contains the last cycles executed before the trigger is fired. **Note:** Some of the pre-trigger memory will not be used if the allocated pre-trigger memory is greater than the collected data (when the trigger occurs early in the run).
When trigger occurs	The trigger does not stop the emulation. Trace switches to post-trigger memory, even if pre-trigger memory is not full. When the specified amount of post-trigger memory is full, the run continues, but no more data is stored.

Example: In the following example, static trigger `tgram` is used to switch from pre-trigger memory to post-trigger memory.

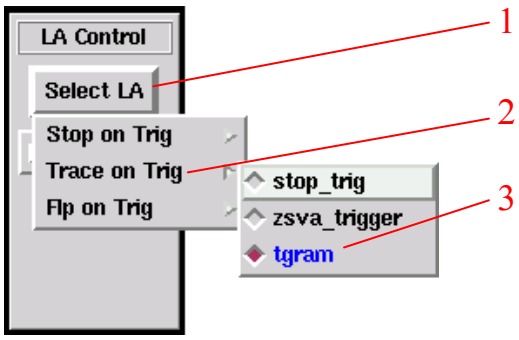


Figure 26: *LA Control → Trace on Trig* **menu item**

1. Click **Select LA**.
2. From the pop-up menu, click on **Trace on Trig**.
3. Select a trigger name.

When you have selected the trigger, the **Trace Panel** is displayed.

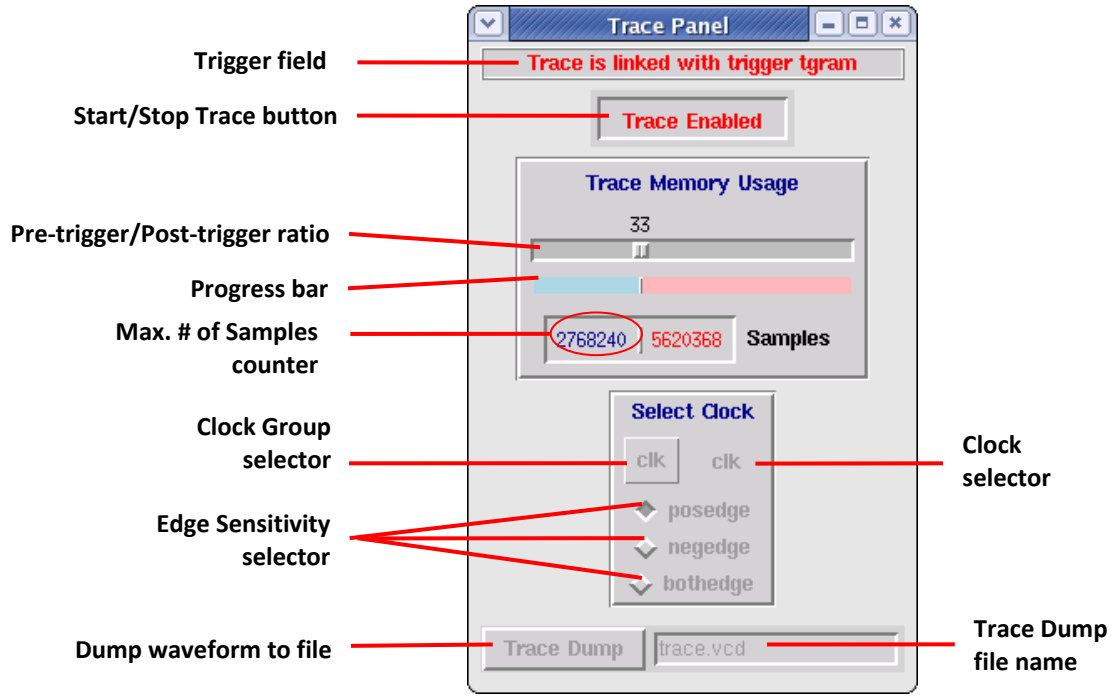


Figure 27: *Trace Panel* **for** *Trace on Trig*

The **Start/Stop Trace** button has two states:

- **Start Trace**
- **Trace Enabled**: when trace is running, click this button to stop it.

Trace Memory Usage frame:

- **Pre-trigger/Post-trigger** ratio: use slider to vary percentage of trace memory.
- **Pre-trigger Progress** bar (light blue): progress is shown in dark blue.
- **Post-trigger Progress** bar (light pink): progress is shown in dark pink.
- **Max. # of Samples** counter**:** indicates the maximum number of samples which can be captured and stored, according to the **Pre-trigger/Post-trigger** ratio.

Select Clock frame:

- **Clock Group** selector: click to select a sampling clock group.
- **Clock** selector**:** click to select a sampling clock from the current clock group. This button is grayed out if the group does not contain more than one clock.
- **Edge Sensitivity** selector**:** set one of 3 edge sensitivities for the selected clock.

The **Trace Panel** for *Trace without Trig* is similar to the **Trace Panel** for *Trace on Trig*. The only difference is the **Trigger** field which is empty in case of *Trace without Trig*.

Using the Trace feature is very simple. For instance, you can:

- Click the **Start Trace** button even while emulation is running.
- Run a certain number of cycles (at least one cycle with the selected clock).
- Click the **Stop Trace** button even while emulation is running.
- Store the results in the selected waveform file (`VCD`, `BIN` or `FSDB` format) even while emulation is running.
- Open the waveform file with an appropriate viewer such as GTKWave.

Notes:

- When you select a trigger (Step 3 in Figure 26) the **Trace** panel pops up with Trace already enabled and `driverClk` set as the default clock. You have to stop Trace if you want to select another sampling clock.
- If you use **`zRun`** with `-debugDriverClk` option, you can set `driverClk` as the trigger sampling clock. If so, no edge sensitivity can be selected.

The two different memory areas for pre- and post-trigger have to be defined as a percentage of the available trace memory capacity. In the example of the previous figure, you allocate 50% of the memory to the pre-trigger area and 50% to the post-trigger area (these are the default settings).

The **Trace Dump** button can only be accessed **if at least one step of the selected clock has run**. Note that trace dumping is dynamic even when emulation is running.

3.5.2.2 Default waveform file name and format

Enter a filename in the **Trace Dump** field and dump the traces into the selected waveform dump file in `VCD`, `BIN` or `FSDB` format. The default format is `VCD` and the default filename is `monitor.vcd` (see Note, end of Section 3.3.3).

3.5.2.3 Combining trigger and trace features

It is possible to combine the *Stop on Trig* and *Trace without Trig* features to stop the emulation when the trigger if fired. This is a handy way to obtain a trace in order to understand why the trigger was fired:

1. Click on **Stop On Trig** then select the associated trigger (Section 3.5.1). Note that Trace is enabled by default and `driverClk` is the default clock.
2. Activate the *Trace without Trig* feature (Section 3.3.3).
3. Launch the run and wait until *Stop on Trig* trigger is fireds.
4. Store the Trace and open the waveform file with an appropriate viewer.

3.5.3 Flp On Trig feature (FOT)

3.5.3.1 Usage

The *Flp on Trig* feature can be used to activate dumping from flexible probes on a trigger event. It combines the following:

- Programming of the *Stop on Trig* feature.
- Activation of the *Flp on Trig* feature, after the clock is stopped.
- Automatic restart of the clock (see Section 3.5.3.2).

Notes:

- Advanced debug with flexible probes is described in Section 3.7.
- There is no **zRun** Tcl command integrating the *Flp on Trig* feature, but a Tcl script can be created for an equivalent scenario (see Section 4.7.7).

The *Flp on Trig* feature is "one shot" (which means that flexible probes are no longer linked to a trigger after the trigger is fired). To use the *Flp on Trig* feature again, you must close the **Flexible Local Probes** panel and start a new *Flp on Trig* session. Any trigger declared in the DVE File can be selected from the **Flp on Trig** combo box.

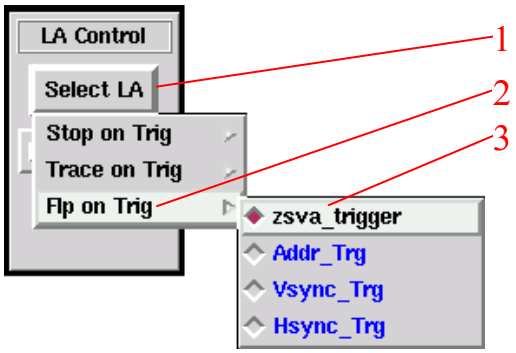


Figure 28: ***LA Control → Flp on Trig*** **menu item**

Before a trigger event (see Figure 29, left).	- The **Flexible Local Probes** panel displays the following message in red: `Flp is linked with trigger <trigger_name>` - The **Waveform Dump** frame is not active (**OFF**). - The **Start Dump** button is disabled. - The **Flush** button is disabled.
After a trigger event (see Figure 29, right)	- The **Flexible Local Probes** panel no longer displays the previous message (no more link with trigger). - The **Waveform Dump** frame becomes active (**ON**). - The **Stop Dump** button is enabled (replaces **Start Dump**). - The **Flush** button is enabled.

By default, all the flexible probe groups displayed in the list of **Enabled groups** and `driverClk posedge` are selected. The **zRun** GUI allows you to make additional changes to the selection such as:

- Unselecting one or more flexible probe groups (see Section 3.7.1).
- Selecting another clock combination (see Section 3.7.2).

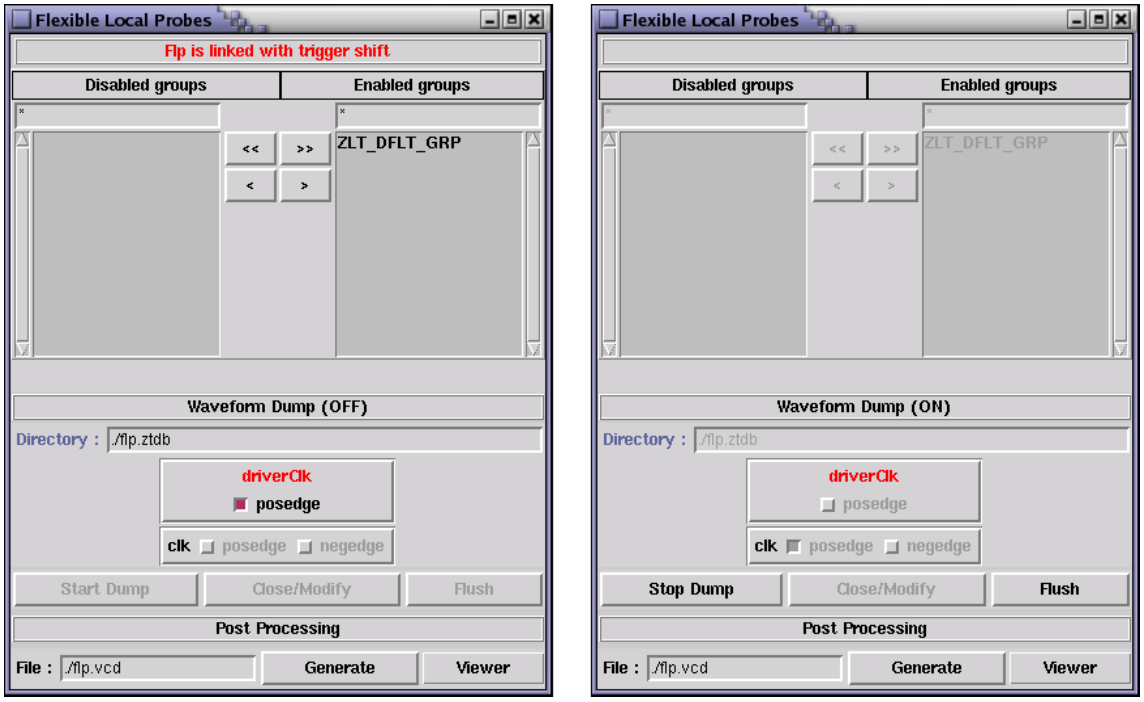


Figure 29: ***Flexible Local Probes*** **panel (before/after a trigger event)**

3.5.3.2 Clock behavior during Flp On Trig sessions

The clock modes (Free/Run/Step) are always accepted during ***Flp on Trig*** sessions. When the trigger fires, the clocks are stopped, the flexible probes are triggered, and the clocks are automatically restarted.

3.6 Memory Read/Write operations during Emulation

You can use the **Memory Hierarchy Browser** from the **Memory Control** panel to read/modify the contents of the following memories in your design:

- `zMem`-generated memories: `zMem` instances containing BRAMs, RAMLUTs, or zrm memories.
- User-defined memories: BRAM instances, RAMLUT instances, zrm instances.

3.6.1 Memory Control panel

The **Memory Control** panel is only available if the design uses memories. You can initialize all memories when you open your session, according to the optional memory file declaration in the `designFeatures` file.

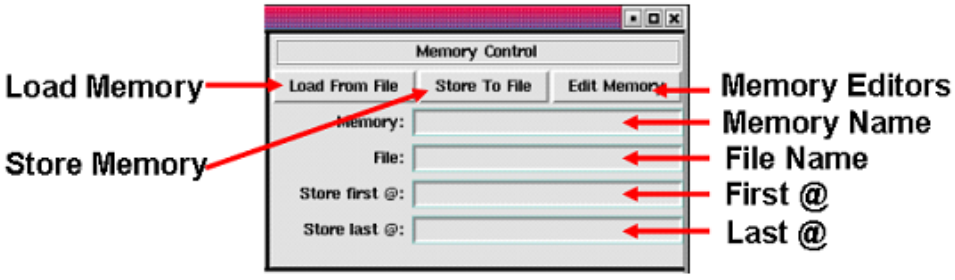


Figure 30: ***Memory Control*** **panel**

The top part of the panel has 3 buttons:

Load From File	Reads the contents of a file and loads it to the current memory.
Store To File	Writes the contents of the current memory into a file; this opens a file browser where you select the path to store the file and enter the filename, or select an existing filename to overwrite with the new data.
Edit Memory	Opens the **Memory Hierarchy Browser** (see Section 3.6.2).

The lower part of the panel has 4 fields for **Load From File/Store To File** operations:

Memory	Selected memory name with the full hierarchical path to the memory in the design that will be used for store and load operations. Right-click this field to open the **Memory Hierarchy Browser** (see Section 3.6.2).
File	Name of the file to be loaded into memory or to be filled with memory content. The file format is described in the ***ZeBu Co-Simulation Manual*** and the ***ZeBu Reference Manual***. Right-click this field to open the file browser.
Store first @ / Store last @	For store operations only, first and last memory addresses to be written to (not used for load operations).These fields are optional but, if one is given, the other must be given as well; if none is given, the complete memory is written to the file.

3.6.2 Memory Hierarchy Browser

Use the **Memory Hierarchy Browser** to select a memory for store/load operations.

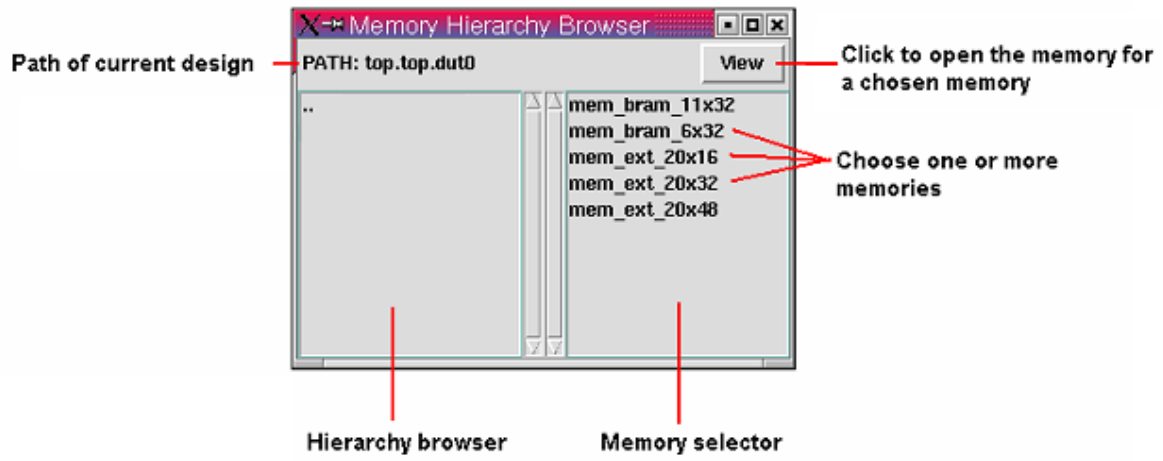


Figure 31 : ***Memory Hierarchy Browser***

The memory hierarchy browser contains the following:

PATH	Indicates the current path in your design with an extra top level that encompasses all of your hierarchy.
Hierarchy browser (left pane)	Displays the memory hierarchy in the design.

Memory selector (right pane)	Displays the memories available in the current level of the hierarchy so that you can choose the memory that you want to load or store.
View button	Adds the selected memory to the **Memory** field in the **Memory Control** panel. If you click **View** when a hierarchy tree is selected in the hierarchy browser pane, the last memory of the tree is selected.

To edit a single memory: Click the memory name and click **View**, or double-click the memory name. This opens the memory contents table for the selected memory.

To edit several contiguous memories: Click on the first memory name, hold down the SHIFT key and click on the last memory name you want to select, then click **View**. This opens the memory contents' tables for the selected memories.

To select several non-contiguous memories: Click on one memory name, hold down the CTRL key and click on each memory name you want to select, then click **View**. This opens the memory contents' tables for the selected memories.

To edit every memory in a hierarchy tree: Select a tree in the hierarchy browser pane and click **View**. This opens the memory contents' tables for all memories in the selected hierarchy level.

3.6.3 Memory Contents table

3.6.3.1 Description

You can view or edit each memory of the design. If you select several memories to edit, there will be one **Memory Contents** table opened for each selected memory.

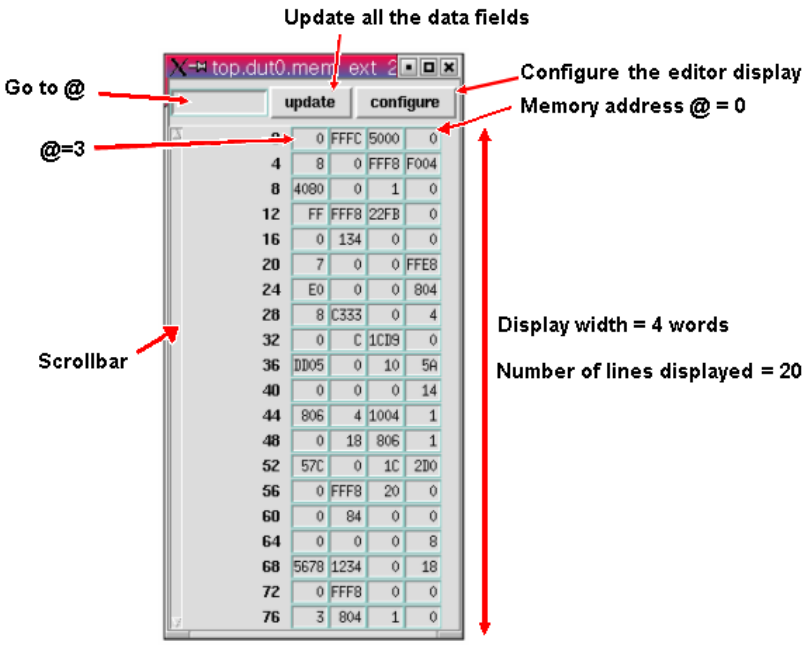


Figure 32: *Memory Contents Table*

Scrollbar	For large memories, this allows you to reach any address.
Go to @	Address selector field; enter the address number **in decimal** in this field (use `0x` prefix for hexadecimal address numbers). The table will update and display your chosen address at the top of the table, for example if you enter 84 in the address field (**Go to @**) and click **Update**, the display will show lines 84 to 160.
Update	Click on **Update** to force the display with current memory values. This feature is also available while the design is running (when the clocks are in **Run** or in **Free run** mode).
Configure	Click on **Configure** to edit the memory contents table.

Notes:

- zrm and BRAM fields are in black and blue, respectively.
- The display for external zrm memories is updated AUTOMATICALLY when an emulation is stopped.
- The display for BRAM memories is NEVER updated automatically, neither when you open a BRAM memory for the first time nor when the emulation is stopped. You must therefore click **Update** except when the memory contains a BRAM/RAMLUT and one of these clocks is declared in `designFeatures`:
  - Free-running Smart Z-ICE/Direct ICE target clock (`zIceClockPort`)
  - Free-running controlled clock (`zceiClockPort`)
  - Synthesized clock (`zClockPort`)

3.6.3.2 Configuration

Click **Configure** to open the **Memory Editor**. Each **Memory Contents Table** has its own **Memory Editor**.

By default each memory editor displays 4 words per line (each word has design memory bit length) and 20 lines. It is possible to configure the line display bit by bit.

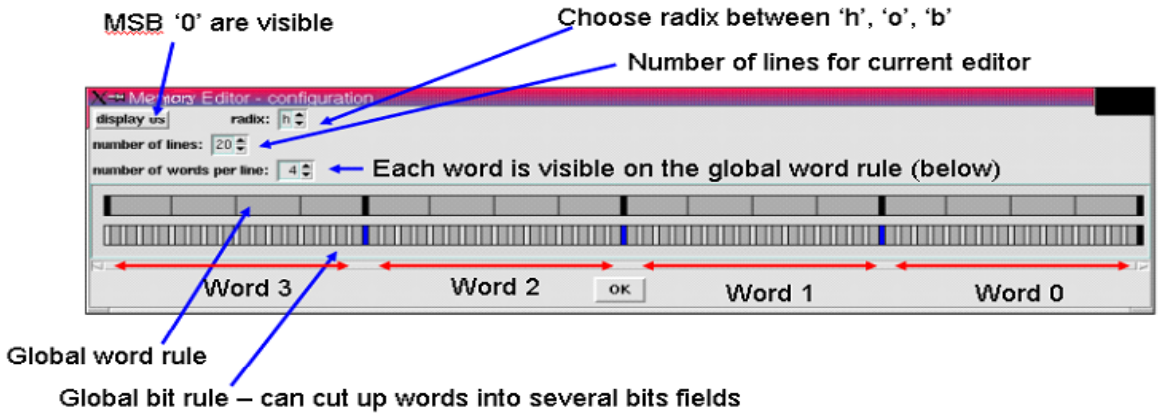


Figure 33: *Memory Editor*

Display 0s	Displays the MSB 0's in each memory editor field (`disabled` by default). When `disabled`, a value of 0010 is shown as 10.
Radix	Is the radix selector, you can choose to display the memory contents in hexadecimal (h), octal (o) or binary (b) format.
Number of lines	Configures the number of lines that are displayed in the memory contents table. **Note:** If the total number of lines in the memory is greater than the selected number, a scrollbar will appear in the dialog box.
Number of words per line	Sets the number of word cells displayed on each line.
Global word rule	Reference rule for words; you cannot edit this rule, but you use it as a reference when editing the **global bit rule**. Each word is 16 bits.
Global bit rule	Used to subdivide each word into several fields. Click on the bit separators in the **Global bit rule** to split the words into smaller bit-lengths. This is useful when each word contains several types of information represented by data of smaller bit-lengths.

3.6.3.3 Example of Memory Editor usage

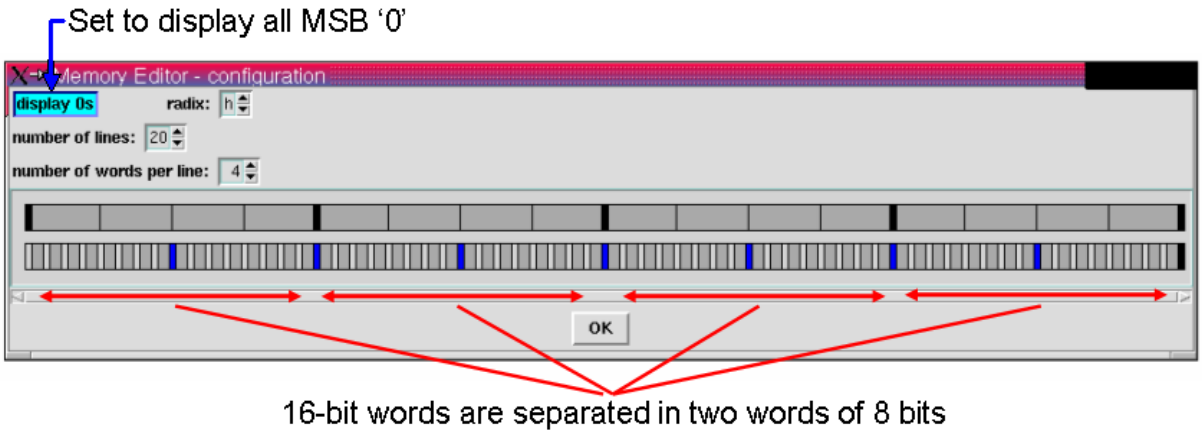
To open the **Memory Editor** panel:

1. In the **Memory Control** panel, click **Edit Memory**.
2. In the **Memory Hierarchy Browser**, select a memory name and click **View**.
3. In the **Memory Contents** table, click **Configure**.

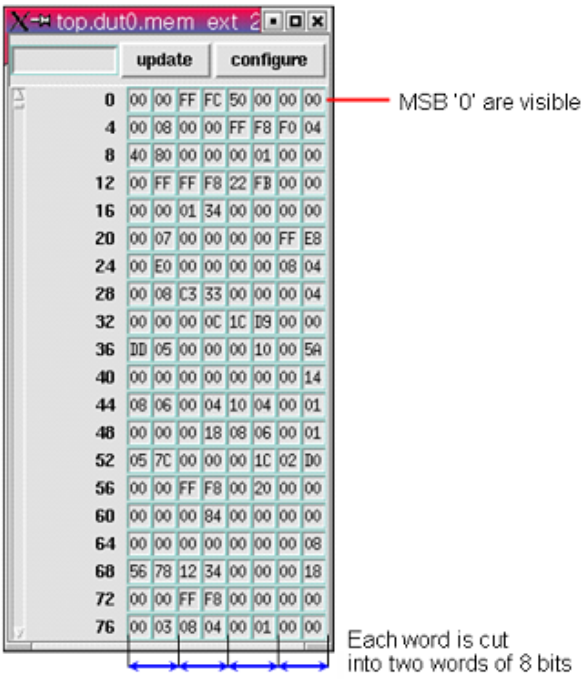
To use the **Memory Editor**:

1. Specify whether MSB `0`s will be shown and select **radix** for displayed values.
2. Select max. number of **lines** that will be displayed and number of **words/line**.
3. Split the words by required **bit length.** Click on appropriate bit separators.

In the following figure, the **Memory Contents** table will display all 0s and a maximum of 20 lines. It will also display eight 8-bit words on each line.



Each 16-bit word is split into two 8-bit words, thus there will be a total of 8 cells per line in the resulting table.



3.7 Advanced debug with Flexible Probes

Flexible Probes increase the debug capabilities while maintaining high performance.

Flexible Probes do not impact the emulation performance except when they are actually enabled. However, Flexible Probes have less impact on speed than Dynamic Probes. Flexible Probes can be declared for modules, instances, signals or ports. They are declared on a group basis for easier runtime activation (refer to ***ZeBu Compilation Manual*** for more details).

The maximum number for Flexible Probes is much higher (more than 30,000 per FPGA) than for static probes, but adding too many Flexible Probes impacts the FPGA filling rate.

The **zRun** emulation interface provides easy runtime control of the Flexible Probes declared at compilation:

- Per-group basis activation
- Waveform dump

The **Flex Probes** button of the **zRun Global Command Panel** opens the **Flexible Local Probes** panel. It can only be used when the connection with ZeBu is open and if at least one group of Flexible Probes has been declared in the DVE file. To enable Flexible Probes in your design when the **zRun** GUI is active:

1. If connection to the ZeBu system is closed, click **Open** to connect.
2. Click **Flex Probes** to display the **Flexible Probes** panel.
3. Once in the **Flexible Probes** panel, choose to enable/disable one or more groups.

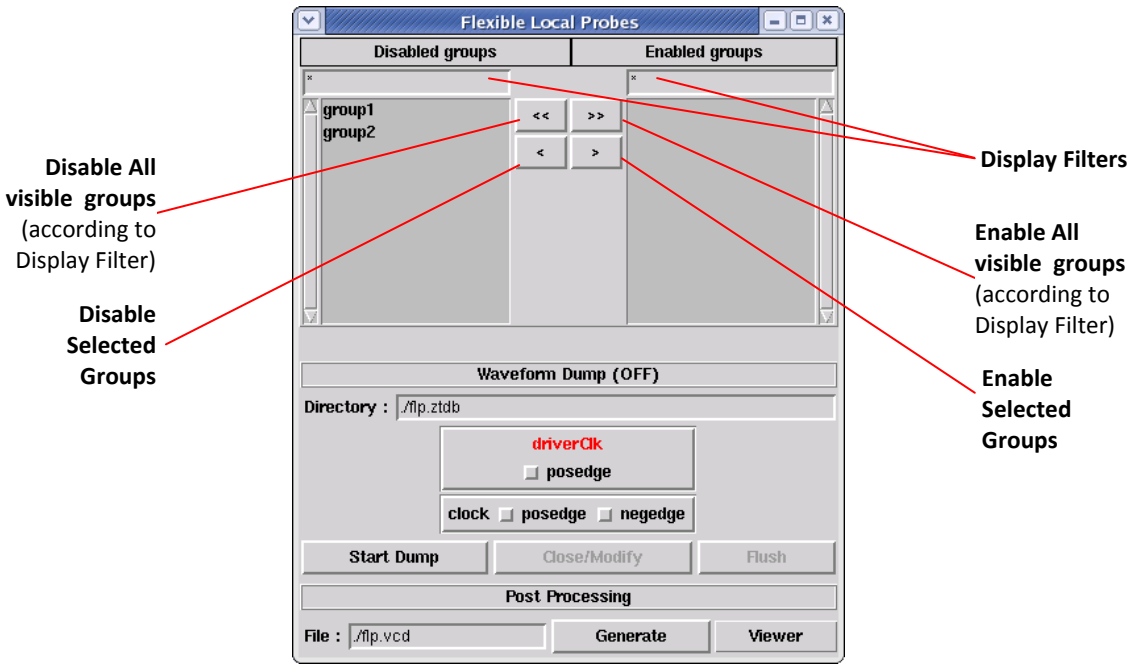


Figure 34: *Flexible Local Probes* panel (No selection)

3.7.1 Enabled Groups and Disabled Groups frames

The **Disabled Groups** pane (left-hand pane) lists the groups that were declared at compilation. You can enable as many disabled groups as you wish. To enable one or more groups, use the appropriate button (>> to enable all the displayed groups, considering the display filter, or > to enable only the selected groups).

The **Enabled Groups** pane (right-hand pane) lists the groups that are currently active. To disable one or more groups, use the specific display filter to narrow your selection and use the appropriate button (<< to disable all the displayed groups, considering the display filter, or < to disable only the selected groups).

Usual hot keys (SHIFT+LeftClick and CTRL+LeftClick) can also be used for selection.

3.7.2 Waveform Dump frame

You can only start dumping a waveform for the selected flexible probes and/or modify the settings for these probes when ALL the clocks are stopped.

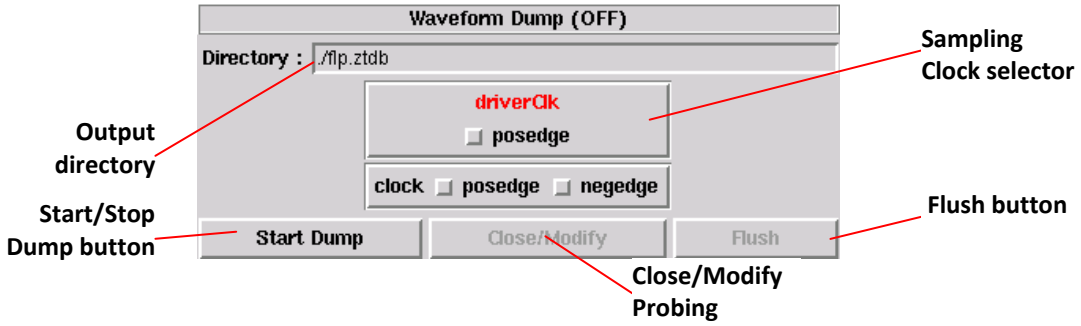


Figure 35: ***Flexible Local Probes → Waveform Dump*** **frame**

Output Directory	Indicates the output directory where the waveform dump will be stored. When the output directory is not grayed out, you can edit it or right-click the field to open a directory browser.
Sampling Clock selector	Selects a different sampling clock via a **Group→Clock→Edge** drop-down menu.
Start/Stop Dump	Starts/stops waveform dumping in specified output directory.
Close/Modify	After the waveform dump is stopped, definitely closes the waveform dump session to allow modification of parameters such as probe groups, sampling clock, and output directory.
Flush	During the waveform dump, flushes the waveform into the specified dump file without closing it. This is useful when you use the **nWave (streamed)** mode so that the nWave viewer updates its content accordingly.

3.7.3 Post Processing frame

The **Post Processing** frame enables conversion of the binary dump file into a `.vcd` or

`.fsdb` file, according to the viewer (`.vcd` for GTKWave; `.fsdb` for nWave).

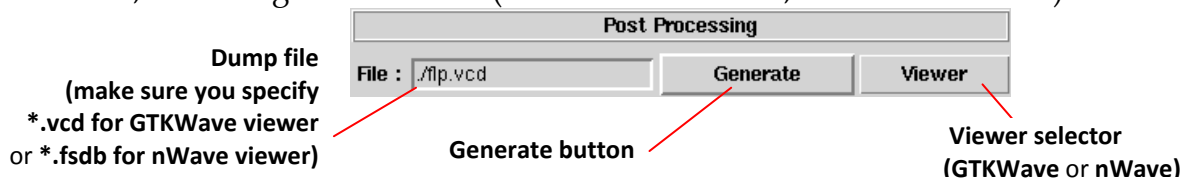


Figure 36: ***Flexible Local Probes → Post Processing*** **Frame**

File Specify in this field a name and an extension for the output dump file, in `VCD` or `FSDB` format. A right click on this field opens a file browser.

Generate Starts the background conversion of the dump file with **`ztdb2vcd`** or **`ztdb2fsdb`** tool, according to the type of the output dump file.

Viewer Click this button to select a waveform viewer via a drop-down menu. Make sure you specify a `.vcd` file extension for your dump file before you select the GTKWave viewer, or a `.fsdb` file extension before you select the nWave viewer.

Note: Post-processing is a background operation which should be launched only when the dump is stopped to avoid any conflict with other operations (in particular if the user sends a new **Generate** or **Viewer** request during post-processing).

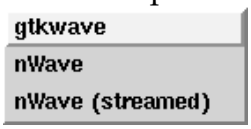


Figure 37: ***Waveform Viewer*** **selector**

Note: The **nWave (streamed)** mode is not available for post-processing in V6_3_0.

3.7.4 Examples

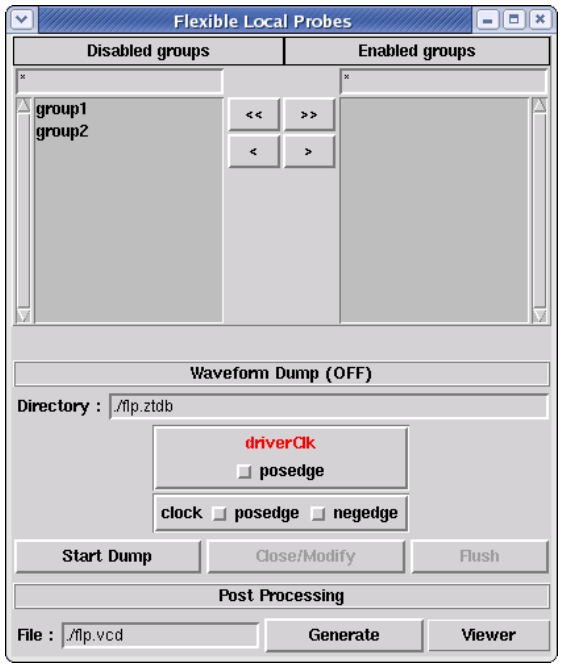


Figure 38: Flexible Probing OFF (no sampling clock, no groups enabled)

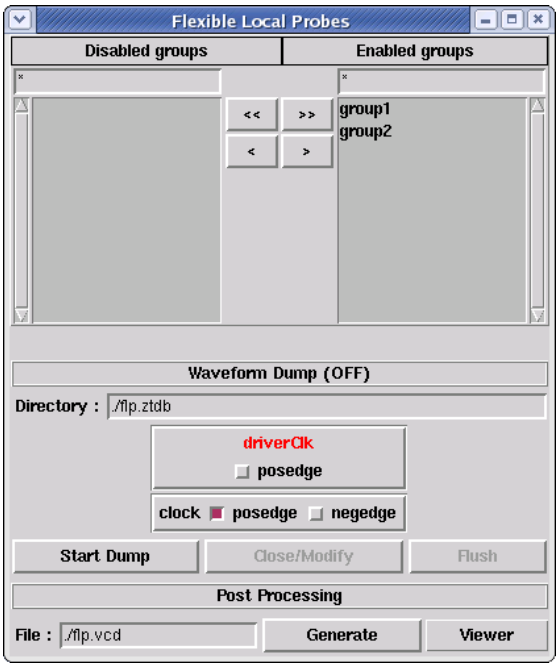


Figure 39: Flexible Groups enabled

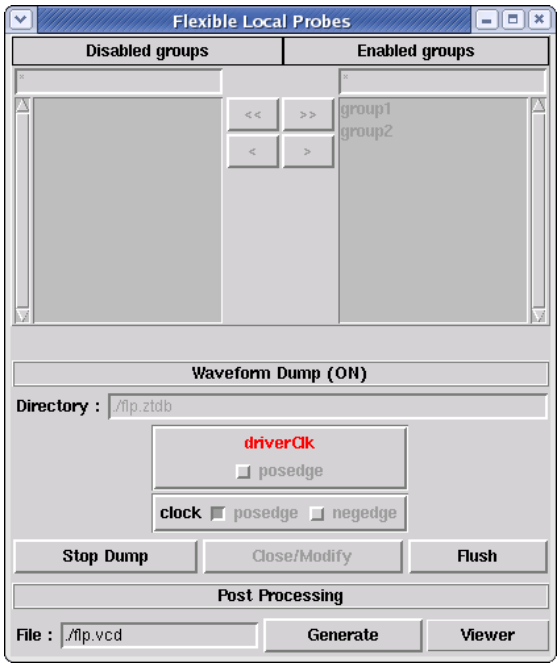


Figure 40: Flexible Probing started

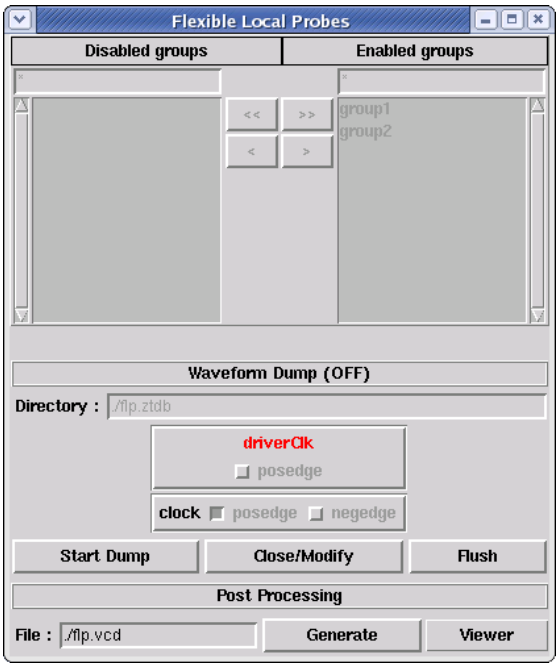


Figure 41: Flexible Probing stopped

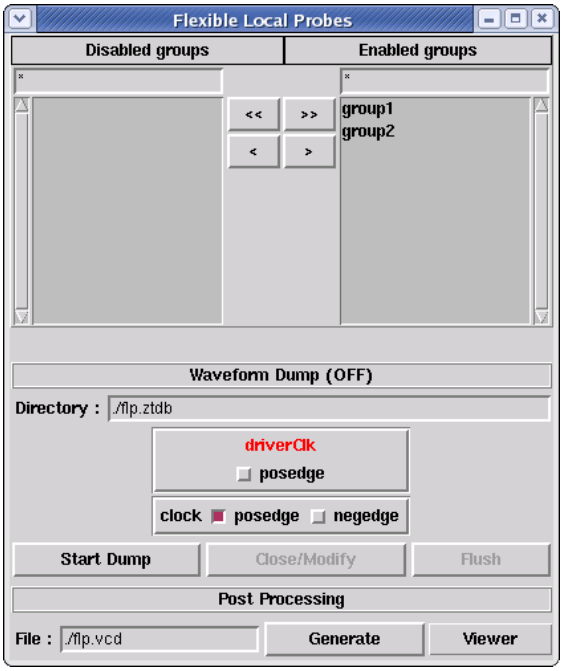


Figure 42: Flexible Probing closed

3.8 ZEMI-3 Debug Monitor

zRun graphical interface includes specific features for control and debug of ZEMI-3 transactors. In the **zRun Global Command** Panel, the **Zemi 3** button opens the **ZEMI-3 Debug Monitor**:

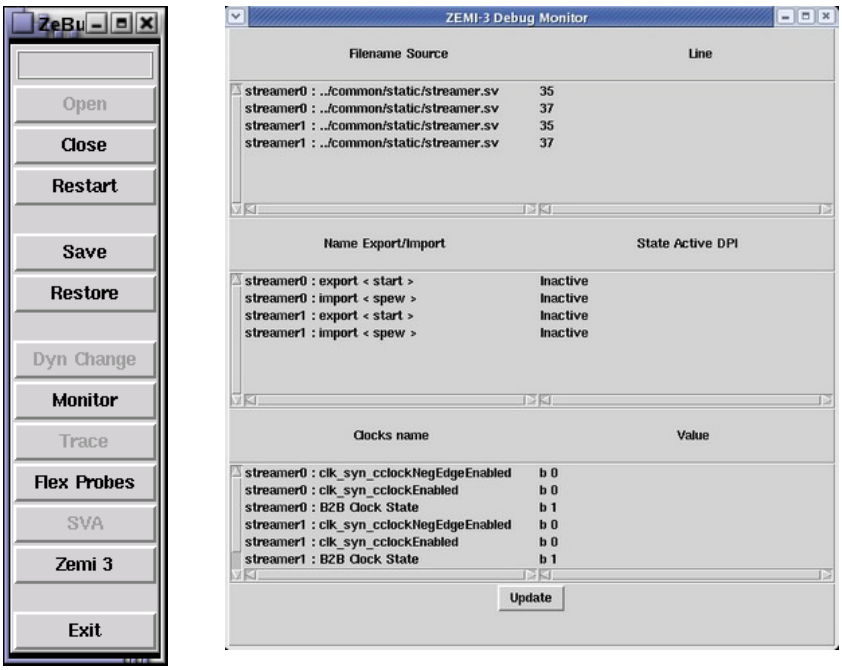


Figure 43: *ZEMI-3 Debug Monitor*

3.8.1 Filename Source frame

The **Filename Source** frame shows the current transactor status with respect to the source file.

Syntax:

```
<transactor_instance_name> : <path/name_of_source_file> <line_number>
```

Example:

```
Streamer0 : ../common/static/streamer.sv   35
```

3.8.2 Name Export/Import frame

The **Name Export/Import** frame lists the status of each C-call DPI function used:

Syntax:

```
<transactor_instance_name> : <function_type> <function_name> <function_status>
```

Example:

```
Streamer0 : export <start>   Inactive
```

The function status is either of the following:

- `Inactive`: This `DPI` function has not been called and has not been executed.
- `Inactive | Wait for HW to run export call`: This DPI export function will not be active until another DPI function is inactive.
- `Active | Wait for SW to finish import call`: This happens if several other DPI calls are running at the same time.
- `Active | Running`: The DPI function is under execution.
- `Active | Wait for SW to accept return value`: The DPI export function is waiting for a value to be returned by the DPI software part (the DPI export function is waiting for a software import or another software function to be finished before a value is returned). This status only occurs if the streaming option is not used .

3.8.3 Clocks Name frame

The **Clocks Name** frame contains information on clock status and back2back status:

3.8.3.1 Clock status

The information is given for positive edge (`clockEnabled`) and for negative edge (`clockNegEdgeEnabled`).

Syntax:

```
<transactor_instance> : <clockname>_<clocktype>_cclockEnabled   b [0 or 1]
<transactor_instance> : <clockname>_<clocktype>_cclockNegEdgeEnabled   b [0 or 1]
```

Example:

```
Streamer0 : clk_syn_cclockEnabled   b 0
Streamer0 : clk_syn_cclockNegEdgeEnabled   b 0
```

3.8.3.2 Back2Back clock status

The information on back2back clock status is only given when the transactor uses at least one export function.

Syntax:

```
<transactor_instance_name> : B2B Clock State   b [0 or 1]
```

Example:

```
Streamer0 : B2B Clock State   b 1
```

The previous figure shows an example used on two ZEMI-3 transactors containing C-call import and C-call export DPI functions (HW/SW) for each transactor.

- If the export function is not called, the clock of the transactor does not run and the value is `0`.
- If the export function is under execution in the hardware or if the `back2back` option is is disabled.

3.9 SystemVerilog Assertions panel

The **Global Command** panel includes an **SVA** button which is active when some SystemVerilog assertions have been synthesized in your design. It opens the **System Verilog Assertion** panel:

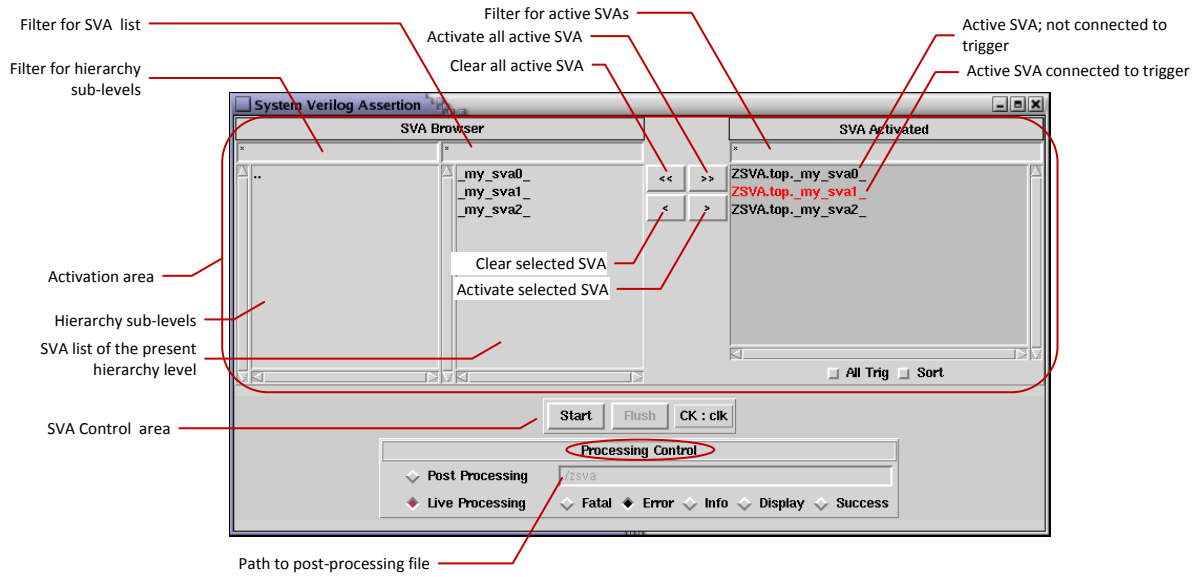


Figure 44: *SystemVerilog Assertion* panel

The settings of this panel are defined before starting the emulation. They cannot be modified when the emulation is running.

The **Processing Control** frame gives access to the following settings:

- Processing mode:
  - **Post Processing**: the state of assertions is stored during emulation in a binary file for post-emulation analysis. The path to the file can be set in the dedicated field.
  - **Live Processing**: the messages for failing assertions are displayed.
- Filter for the reported assertions (for **Live Processing** only):
  - **Fatal**: displays only information for assertions which have an action block with `$fatal` are displayed.
  - **Error**: displays only information for assertions which have an action block with `$error` and `$fatal` or assertions without any action block.
  - **Info**: displays only information for assertions which have an action block with `$info`, `$error` and `$fatal` or assertions without any action block.
  - **Display**: displays only information for assertions which have an action block with `$display`, `$info`, `$error` and `$fatal` or assertions without any action block.
  - **Success**: all assertions are reported, including successful assertions.

In the activation area of the panel, user can select which of the synthesized SystemVerilog Assertions will be active at runtime. A display filter is available for each list of the panel; all items or only the selected items of the frame can be activated/disabled with the appropriate buttons of the panel.

The SVA control area has 3 buttons:

- **Start**: Start SVA analysis or SVA post-process file dumping.
- **Flush**: Displays all the SVA report messages that are already generated by the hardware but not yet displayed by the software.
- **CK**: Selection of the reference clock.

3.10 Tcl Command field

3.10.1 Description

zRun graphical interface is written in Tcl and supports Tcl commands entered by user in the **Tcl command** area as displayed on Figure 45.

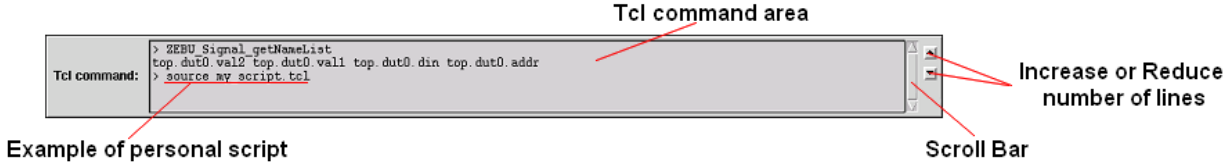


Figure 45: *Tcl Command* field

Any Tcl command is valid including the **zRun**-specific commands described in Chapter 4 and you can execute Tcl scripts using the Tcl `source` command.

On the right side of the **Tcl command** field, a scroll bar is available to see the history of the commands and results. You can reduce or increase the number of lines in the **Tcl command** area using dedicated ARROW buttons.

The **Tcl command** area works as a simplified Shell display:

- UP and DOWN keys give access to the command history (20 previous commands available)
- HOME and DELETE keys operate as in standard Shell.
- Mouse Copy/Paste: LEFT button for **Copy** and RIGHT button for **Paste**, from the same window or from another one.

Note: When executing Tcl commands or scripts, the other panels of the **zRun** GUI are not updated according to the results.

3.11 Tcl log file

zRun GUI creates a file that contains all the commands that have been implicitly and explicitly entered through the interface. This file is a Tcl script that can be used in 3 ways:

- In batch mode with the `-nogui` and `-do` options (see example below)
- In pre-open GUI session (`init` for instance) with the `-do` option
- As a script during a GUI session (for example `my_zRun_cmds.tcl`)

Note: This log file does not include the commands typed in the **Tcl command** field.

The generated Tcl script is called `zRun_cmds.tcl`. It is located in the same directory where **zRun** was launched.
Make sure you rename this file. It is overwritten each time a new run begins.

Example: Using the renamed `my_zRun_cmds.tcl` file to run **zRun** in batch mode:

```
$> zRun -design design.setup -nogui -do my_zRun_cmds.tcl
```

Note: In the example above, remove the `-nogui` option to run **zRun** in GUI mode.

4 zRun Tcl Commands

When launching **zRun**, you can send commands through a personal Tcl script using the `-do <my_tcl_script>` option in the command line. The Tcl script can contain any Tcl command, as well as the ZeBu specific commands listed in this chapter.

You can also execute a Tcl script with **zRun** in batch mode if you use the `-nogui` option in the **zRun** command line or by sourcing the script in the **Tcl Commands** field in the **zRun** GUI as described in Section 3.9.

4.1 Connecting the ZeBu system

4.1.1 Opening a Session when zRun is Launched

When **zRun** is launched, no command is necessary at the beginning of a Tcl script to open a session since **zRun** initializes ZeBu before it executes any scripts.

4.1.2 Opening a Session after a ZEBU_Close

`ZEBU_open` initializes ZeBu. This command is necessary after a `ZEBU_close` when the emulation session needs to be restarted.

Syntax:

```
ZEBU_open <pid>
```

Note: If a testbench has been specified in the **zRun** command line, the `ZEBU_open` runs this testbench until the initialization step. If no testbench has been specified (stand-alone design or embedded testbench), a default session is started. In both cases, the design is loaded and the board configured (including the clocks).

When **zRun** is used with the `-do` option and a Tcl script, the `ZEBU_open` call is executed during the first call to any `ZEBU_<xxx>` function. This means that Tcl commands executed before the first call to a `ZEBU_<xxx>` function are executed before a connection is made to the machine.

4.1.3 Closing a session

`ZEBU_close` closes a session, after all the commands for this session have been executed. You can open a new session after the previous one has been closed.

Syntax:

```
ZEBU_close
```

Note: Calling `ZEBU_close` then `ZEBU_open` can be used to simulate a restart.

4.1.4 Getting a session status

`ZEBU_getStatus` retrieves the current status of **zRun** (already connected to ZeBu or not, `ZEBU_open` already executed or not). Returns `open` or `close`.

Syntax:
```
ZEBU_getStatus
```

Example: Display the current **zRun** status
```
puts "Currently zRun status: [ZEBU_getStatus]"
```

4.1.5 Exiting a session

After all the tests have been performed, use the `ZEBU_exit` command to exit **zRun**.

Syntax:
```
ZEBU_exit <exitCode>
```

4.2 Running the design verification

Once the session is open you can execute the design verification. It is forbidden to call any of the following commands if ZeBu has not been correctly opened.

4.2.1 Getting the Clock Groups list

In most cases, only one clock group (default group) is used. But it is possible to define several groups in the `designFeatures` file with a `GroupName` option. Each group is asynchronous with respect to other groups and can contain several clocks.

`ZEBU_Clock_getGroupNameList` returns a list of defined clock groups. Each name in this list can be used to get the list of clocks in the group.

Syntax:
```
ZEBU_Clock_getGroupNameList
```

Example: Display all clock groups:
```
# get the list of all clock groups
set clkGroupList [ZEBU_Clock_getGroupNameList]
# display it
foreach group $clkGroupList { puts "Clock Group: $group" }
```

4.2.2 Getting the Clock list

`ZEBU_Clock_getNameList` returns the list of clocks for a specific clock group.

Syntax:
```
ZEBU_Clock_getNameList <clk_group>
```

Where: `<clk_group>` is the group name returned by `ZEBU_Clock_getGroupNameList`

Example: Display a list of all groups and all clocks in each group:
```
set clkGroupList [ZEBU_Clock_getGroupNameList]
# loop on each group
foreach group $clkGroupList {
  # display the group name
  puts "Clock Group : $group"
  # get the list of clocks in the group
  set clockList [ZEBU_Clock_getNameList $group]
  # display every clock of the group
  foreach clock $clockList { puts "clock $clock" }
}
```

4.2.3 Enabling clocks for N cycles

Only one clock name of a clock group is necessary to enable the group as a whole. When the group is enabled, all the clocks of the group will run until the specified clock has run for a given number of cycles. Each group is controlled separately; when the goal is to run all groups concurrently, each of them must be enabled.

If more than one clock exists in a clock group, the other clocks will run synchronously according to the frequency and waveform given in the clock parameter file. The maximum number of clock cycles ranges into the hundreds of billions of cycles.

Syntax:

```
ZEBU_Clock_enable <clk_name> <nb_cyles>
```

Where: `<clk_name>` is a clock name returned by `ZEBU_Clock_getNameList`
`<nb_cycles>` is the expected number of cycles for the specified clock

Example: Run 10 cycles on each clock group, driven by the first clock of the group.

```
set clkGroupList [ZEBU_Clock_getGroupNameList]
# loop on each group
foreach group $clkGroupList {
  # get the list of clocks in the group
  set clkList [ZEBU_Clock_getNameList $group]
  # get the first clock of the group
  set firstClk [lindex $clkList 0]
  # run 10 cycles on the first clock
  # the other clocks of this group follow synchronously
  ZEBU_Clock_enable $firstClk 10
}
```

Notes:

- This command enables to run more than one group at the same time. Note also that it is not possible to control precisely the start of two groups since there is no link between groups.
- When dumping is enabled, the `ZEBU_Clock_enable` command will not return anything until the run is finished. This limitation does not allow you to run many groups in parallel before dumping is through.

4.2.4 Enabling clocks in Free running mode

`ZEBU_Clock_enableForever` enables a clock without a precise number of cycles to run. This is useful when the ZeBu system is used for software debug.

Syntax:

```
ZEBU_Clock_enableForever <clk_name>
```

Where: `<clk_name>` is a clock name returned by `ZEBU_Clock_getNameList`

Note: With this type of clock enable command, you can start all clock groups simultaneously, using the `-debugDriverClk` option in the **`zRun`** command line. It adds a new special clock group called `driverClk`, which contains only one clock called `driverClk` (as the clock group) and drives the ZeBu system clock:

- Using `ZEBU_Clock_enableForever` on one clock of each sampling clock group: no clock will start physically until `driverClk` starts.
- Using `ZEBU_Clock_enableForever` on `driverClk` makes all clocks of the design start simultaneously.

Example: Start all design clocks simultaneously with `driverClk.`

```
# NOTE: only works if -debugDriverClk is used in the zRun command line
set clkGroupList [ZEBU_Clock_getGroupNameList]
# loop on each group
foreach group $clkGroupList {
  # get the list of clocks in the group
  set clkList [ZEBU_Clock_getNameList $group]
  # get the first clock of the group
  set firstClk [lindex $clkList 0]
  # Start free running mode on the first clock
  # note that no clock starts physically
  # other clocks of this group follow synchronously when started
  ZEBU_Clock_enableForever $firstClk
}
# Start driverClk that physically start all other clocks free running
ZEBU_Clock_enableForever driverClk
```

4.2.5 Disabling clocks

A clock group which has been enabled using `ZEBU_Clock_enable` or `ZEBU_Clock_enableForever` can be disabled using the `ZEBU_Clock_disable` command. This command will stop all the clocks in the same clock group as the selected clock. The selected clock can be different from the one used to enable the group.

Syntax:

```
ZEBU_Clock_disable <clockName>
```

Where: `<clk_name>` is a clock name returned by `ZEBU_Clock_getNameList`

Note: It is not possible to control with precision the time when the group will be stopped, because disabling a clock is always asynchronous. This depends on the PC speed and load, the PCI bus load, etc.

4.2.6 Getting the clock status

`ZEBU_Clock_getStatus` indicates whether the clock is enabled or disabled, which is especially useful when you need to know if a run has terminated.

Syntax:

```
ZEBU_Clock_getStatus <clk_name>
```

Where: `<clk_name>` is a clock name returned by `ZEBU_Clock_getNameList`

`ZEBU_Clock_getStatus` returns `running` or `stopped`.

Example: Wait until the run is finished.

```
set clkGroupList [ZEBU_Clock_getGroupNameList]
# get the first clock group
set firstClkGrp [lindex $clkGroupList 0]
# get the list of clocks in the group
set clkList [ZEBU_Clock_getNameList $firstClkGrp]
# get the first clock of the group
set firstClk [lindex $clkList 0]
# Start 1000000 cycles on firstClk
ZEBU_Clock_enable $firstClk 1000000
# Wait until end of run
# Note : use ZEBU_getStatus because a testbench (if used)
# could close the session during the run
while {    [ZEBU_getStatus] == "open" \
        && [ZEBU_Clock_getStatus $firstClk] == "running" } {
#put a message on stdout and wait .5 sec
  puts "$firstClk still running"
  after 500
}
```

4.2.7 Getting the executed clock cycles

`ZEBU_Clock_getCounter` returns the number of cycles a clock has executed since the last `ZEBU_open` command. This command is available during a run.

Syntax:

```
ZEBU_Clock_getCounter <clk_name>
```

Where: `<clk_name>` is a clock name returned by `ZEBU_Clock_getNameList`

Example: Display the number of cycles processed during a run:

```
set clkGroupList [ZEBU_Clock_getGroupNameList]
# get the first clock group
set firstClkGrp [lindex $clkGroupList 0]
# get the list of clocks in the group
set clkList [ZEBU_Clock_getNameList $firstClkGrp]
# get the first clock of the group
set firstClk [lindex $clkList 0]
# Start 1000000 cycles on firstClk
ZEBU_Clock_enable $firstClk 1000000
# Wait until end of run
# Note : use ZEBU_getStatus because a testbench (if used)
# could close the session during the run
while {    [ZEBU_getStatus] == "open" \
        && [ZEBU_Clock_getStatus $firstClk] == "running" } {
  # Display number of cycles executed
  set nbCycles [ZEBU_Clock_getCounter $firstClk]
  puts "$firstClk still running : $nbCycles has been executed"
  after 500
}
```

4.3 Controlling memories

External zrm memories, BRAMs and LUTRAMs can be controlled with Tcl scripts.

You can use the **Memory Hierarchy Browser** from the **Memory Control** panel to read/modify the contents of the following memories in your design:

- **zMem**-generated memories: **zMem** instances containing BRAMs, RAMLUTs, or zrm memories.
- User-defined memories: BRAM instances, RAMLUT instances, zrm instances.

4.3.1 Getting a memory list

ZEBU_Memory_getNameList returns the list of memory names. The names are the hierarchical paths to the memories, as described in the ***ZeBu Co-Simulation Manuals*** and the ***ZeBu Reference Manual***, and can be used in Tcl memory commands. This list includes all zrm memories, BRAMs, and LUTRAMs used by the design.

Syntax:

```
ZEBU_Memory_getNameList [hierarchy_separator]
```

Where: `[hierarchy_separator]` is an optional parameter. By default, this separator is a dot (`.`) or the character defined in **zDbPostProc** (if any), but you can specify another one such as a slash (`/`) or a pipe (`|`). This feature can be used as a work-around in a situation where the instance name of the memories includes the same character as the hierarchy separator.

Example: Display all memories of the design:

```
# get the memories list with '.' as hierarchy separator
set memoryList [ZEBU_Memory_getNameList]

# Display it
foreach name $memoryList { puts "Memory $name" }
```

4.3.2 Checking if a memory is writable

ZEBU_Memory_isWritable is a special command which allows you to check whether or not a memory is writable. It has been deprecated since all memories are writable.

Syntax:

```
ZEBU_Memory_isWritable
```

ZEBU_Memory_isWritable returns `1` if true, `0` otherwise.

Example: Display the memory type for the whole design.

```
# get the memories list with '.' as hierarchy separator
set memoryList [ZEBU_Memory_getNameList]
# Display it
foreach name $memoryList {
  if { [ZEBU_Memory_isWritable] == "1" } {
    puts "$name is ZeBu external zrm memory"
  } else {
    puts "$name is BRAM memory"
  }
}
```

4.3.3 Getting a memory size

`ZEBU_Memory_width` returns the bit length of each word (in decimal).

`ZEBU_Memory_depth` returns the number of words of a memory (in decimal).

Syntax:

```
ZEBU_Memory_width <mem_name>
ZEBU_Memory_depth <mem_name>
```

Where: `<mem_name>` is a memory name returned by `ZEBU_Memory_getNameList`

Example: Get the memory size for the whole design:

```
# get the memories list with '.' as hierarchy separator
set memoryList [ZEBU_Memory_getNameList]

# get width and length
foreach name $memoryList {
  puts "Lengh of $name is [ZEBU_Memory_depth $name]"
  puts "Width of $name is [ZEBU_Memory_width $name]"
}
```

4.3.4 Saving memory contents

`ZEBU_Memory_storeToFile` stores the memory contents into a file. This command is dynamic (available during the run).

`ZEBU_Memory_storeToBuffer` retrieves the memory contents and flushes it in the buffer passed as parameter.

Syntax:

```
ZEBU_Memory_storeToFile <mem_name> <file_name> [first_addr] [last_addr] [mode]
ZEBU_Memory_storeToBuffer <mem_name> [first_addr] [last_addr] [format]
```

Where:	`<mem_name>`	memory name returned by `ZEBU_Memory_getNameList`
	`<file_name>`	is the name of the dump file
	`[first_addr]`	is an optional parameter (decimal value of the first memory address to be written in the `<file_name>` or buffer)
	`[last_addr]`	is an optional parameter (decimal value of the last memory address to be written in the `<file_name>` or buffer)
	`[mode]`	is an optional parameter (to set the format of the output file). `t` = text, `b` = binary. Text mode by default
	`[format]`	is an optional parameter: `%b` for binary, `%o` for octal, `%h` for hex

Note: If `[first_addr]` and `[last_addr]` are both not specified, the entire memory is dumped to the file. It is not possible to specify only one of the two.

Example: Store each complete memory in a different file:

```
# get the memories list with '.' as hierarchy separator
set memoryList [ZEBU_Memory_getNameList]
# store it
set i 0
foreach name $memoryList {
  puts "Store memory $name in file mem_$i"
  ZEBU_Memory_storeToFile $name mem_$i
  incr i
}
```

4.3.5 Loading memory contents

`ZEBU_Memory_loadFromFile` loads the memory contents from a file. This operation is dynamic (available during the run).

`ZEBU_Memory_loadFromBuffer` loads the memory contents from a buffer passed as parameter.

Note: During a run, the instant when memory is loaded is not guaranteed (the run continues during command interpretation), which can lead non-reproducible results.

Syntax:

```
ZEBU_Memory_loadFromFile <mem_name> <file_name>
ZEBU_Memory_loadFromBuffer <mem_name> <buf_name> [first_addr] [last_addr] [format]
```

Where:	`<mem_name>`	is one of the memory names returned by `ZEBU_Memory_getNameList`
	`<file_name>`	is the name of the load file
	`<buf_name>`	is the name of the buffer
	`[first_addr]`	is an optional parameter (decimal value of the first memory address to be written in the buffer)
	`[last_addr]`	is an optional parameter (decimal value of the last memory address to be written in the buffer)
	`[mode]`	is an optional parameter (to set the format of the output file). `t` = text, `b` = binary. Text mode by default
	`[format]`	is an optional parameter: `%b` for binary, `%o` for octal, `%h` for hexadecimal

Example: Initialize every memory with different files:

```
# get the memories list with '.' as hierarchy separator
set memoryList [ZEBU_Memory_getNameList]
# initialize it
set i 0
foreach name $memoryList {
  puts "Initialize memory $name with file mem_$i"
  ZEBU_Memory_loadFromFile $name mem_$i
  incr i
}
```

4.3.6 Writing a memory word

`ZEBU_Memory_writeWordBuffer` writes a value in binary format at one valid memory address. The WRITE is dynamic (available during the run).

Notes:

- Do not click **Update** when the memory contains a BRAM/RAMLUT and one of these clocks is declared in `designFeatures`:
  - Free-running Smart Z-ICE/Direct ICE target clock (`zIceClockPort`)
  - Free-running controlled clock (`zceiClockPort`)
  - Synthesized clock (`zClockPort`)
- During the run, the instant when the memory word is written is not guaranteed (the run continues during command interpretation), which can lead to a non-reproducible result.

Syntax:

```
ZEBU_Memory_writeWordBuffer <mem_name> <addr> <value> [format]
```

Where:
- `<mem_name>` is the memory name returned by `ZEBU_Memory_getNameList`
- `<addr>` is the address to write. Must be less than the memory length
- `<value>` is the data to write. Must have the exact bit length of memory
- `[format]` is an optional parameter(format for operation):
  - %b for binary
  - %o for octal
  - %d for decimal
  - %h or %x for hexadecimal (default)

Example: Write `0x35` in first 5 addresses of each on-board memory in the design:

```
# get the memory list
set memList [ZEBU_Memory_getNameList]
for each name $memList {
  # Is it an on-board memory
  if { [ZEBU_Memory_isWritable $name] } {
    # Get exact bit length of the memory
    set width [ZEBU_Memory_width $name]
    # Check if memory has enough bit length
    if { $width < [string length "110101"] } {
      puts "$name : bit length too short to write 0x35"
    } else {
      for {set i 0} {$i<5} {incr i} {
        # Write 0x35 at address i
        ZEBU_Memory_writeWordBuffer $name
                                    $i [format "%0${width}s" 110101] "%b"
      }
      # Open the memory Editor of this memory (gui mode only)
      catch {ZEBU_Memory_Editor $name}
    }
  }
}
```

4.3.7 Reading a memory word

`ZEBU_Memory_readWordBuffer` returns a value in binary format at a valid memory address. This command does not execute a real memory READ, but just returns the software memory word buffer updated by the `ZEBU_Memory_update` described below.

`ZEBU_Memory_update` updates the full software buffer of a memory. This command is dynamic.

Note: Do not click **Update** when the memory contains a BRAM/RAMLUT and one of these clocks is declared in `designFeatures`:

- Free-running Smart Z-ICE/Direct ICE target clock (`zIceClockPort`)
- Free-running controlled clock (`zceiClockPort`)
- Synthesized clock (`zClockPort`)

Syntax:

```
ZEBU_Memory_update          <mem_name>
ZEBU_Memory_readWordBuffer <mem_name> <addr> [format]
```

Where: `<mem_name>` is the memory name returned by `ZEBU_Memory_getNameList`
`<addr>` is the address to read. Must be less than the memory length
`[format]` is an optional parameter (format for operation):

- %b for binary
- %o for octal
- %d for decimal
- %h or %x for hexadecimal (default)

Example: Read the five first addresses of each memory of the design.

```
# get the memory list
set memList [ZEBU_Memory_getNameList]
for each name $memList {
  # Update software buffer
  ZEBU_Memory_update $name
  puts "$name : "
  for {set i 0} {$i<5} {incr i} {
    # Read word at address i
    puts "read word $i : [ZEBU_Memory_readWordBuffer $name $i \"%b\"]"
  }
}
```

4.4 Monitoring signals

The signals of the design which are available as dynamic probes, either automatically or by explicit declaration, can be monitored with the commands described in this section.

4.4.1 Getting a signal list

`ZEBU_Signal_getNameList` returns a list of signal names available as dynamic probes. These signal names are complete hierarchical paths and can be used in Tcl signal commands.

Syntax:

```
ZEBU_Signal_getNameList [<path> [<separator> [showNotSelected [filter]]]]
```

Where: `<path>` is the hierarchical path from which the signal list must be returned
`<separator>` is the hierarchy separator character; default is "." or character defined in **`zDbPostProc`** (if any)
`[showNotSelected [filter]]` `0` returns only selected signal names
`1` returns selected and not selected signal names

Example: Display all the signals of the design that are available as dynamic probes.

```
# get the signals list with '.' as hierarchy separator
set signalList [ZEBU_Signal_getNameList]
# display it
foreach name $signalList { puts "Signal $name" }
```

4.4.2 Getting a local node

`ZEBU_Signal_getLocalNode` returns a list of signal names at sub-hierarchy level.

Syntax:

```
ZEBU_Signal_getLocalNode [<path> <separator> [showNotSelected [filter]]]
```

Where:	`<path>`	is the hierarchical path from which the signal list must be returned
	`<separator>`	is the hierarchy separator character; default is `"."` or character defined in **`zDbPostProc`** (if any)
	`[showNotSelected [filter]]`	`0` returns only selected signal names `1` returns selected and not selected signal names

4.4.3 Getting a local instance

`ZEBU_Signal_getLocalInstance` returns a list of signal names at hierarchy level.

Syntax:

```
ZEBU_Signal_getLocalInstance [<path> <separator> [showNotSelected
[filter]]]
```

Where:	`<path>`	is the hierarchical path from which the signal list must be returned
	`<separator>`	is the hierarchy separator character; default is "." or character defined in **`zDbPostProc`** (if any)
	`[showNotSelected [filter]]`	0 returns only selected signal names 1 returns selected and not selected signal names

4.4.4 Getting signal Information

`ZEBU_Signal_getType` identifies the signal type as one of the following:

- `interface`
- `internal`
- `simulated`

`ZEBU_Signal_isWritable` returns `1` if the signal is writable; `0` otherwise.

`ZEBU_Signal_getWidth` returns the number of digits of the specified signal.

Syntax:

```
ZEBU_Signal_getType <name>
ZEBU_Signal_isWritable <signalName>
ZEBU_Signal_getWidth <name> [format]
```

Where: `[format]` is an optional parameter (format for operation):

- `%b` for binary
- `%o` for octal
- `%d` for decimal
- `%h` for hexadecimal (default)

4.4.5 Getting a signal value

`ZEBU_Monitor_refresh` updates the software signals buffer with a snapshot of all signals available as dynamic probes. It is dynamic (available during the run).

`ZEBU_Signal_read` returns a signal value. It does not execute a real signal READ; it returns the software signals buffer updated by `ZEBU_Monitor_refresh`. Note that it is not necessary to execute `ZEBU_Monitor_refesh` before `ZEBU_Signal_read` if no run has been executed since the last time a `ZEBU_Monitor_refesh` was sent.

Syntax:

```
ZEBU_Monitor_refresh
ZEBU_Signal_read <sig_name> [sig_format]
```

Where: `<sig_name>` is a signal name returned by `ZEBU_Signal_getNameList`

`[sig_format]` is a radix (optional parameter) with the following possible values:

- %b for binary
- %o for octal
- %d for decimal
- %h for hexadecimal (default)

Example: Display the value of `top.i1.sig` in hexadecimal format and the values of every available signal in binary format. Note that no run is executed between two `ZEBU_Signal_read` commands, so `ZEBU_Monitor_refresh` is called only once.

```
# update signals software buffer
ZEBU_Monitor_refresh
# simple get hexadecimal value
set hexVal [ZEBU_Signal_read top.i1.sig "%h"]
puts "Signal 'top.i1.sig' value is $hexVal"
# get the signals list with '.' as hierarchy separator
set signalList [ZEBU_Signal_getNameList]
# display all the binary values
foreach name $signalList {
  set binVal [ZEBU_Signal_read $name "%b"]
  puts "Signal '$name' value is $binVal"
}
```

4.4.6 Setting a signal value

`ZEBU_Signal_write` sets the value of a signal available as a dynamic probe. This command does not execute a real signal WRITE, but it just updates the software signals buffer. Remember to send a `ZEBU_Monitor_flush` command after several `ZEBU_Signal_write` commands, especially prior to a run.

Note: Do not click **Update** when the memory contains a BRAM/RAMLUT and one of these clocks is declared in `designFeatures`:

- Free-running Smart Z-ICE/Direct ICE target clock (`zIceClockPort`)
- Free-running controlled clock (`zceiClockPort`)
- Synthesized clock (`zClockPort`)

Syntax:

```
ZEBU_Signal_write  <signal_name> <value> [sig_format]
ZEBU_Monitor_flush
```

Where: `<signal_name>` is a signal name, as returned by `ZEBU_Signal_getNameList`
`<value>` is a new signal value which must be coherent with `[sig_format]`.
If the `<value>` bit length is greater than the signal bit length, then `<value>` will be truncated, otherwise completed with `0s` on MSBs.

Note: The maximum supported length for decimal format is 64 bits.

Example: How to force and check one signal in different formats.

```
# Get the signal list with full hierarchy path
set signalList [ZEBU_Signal_getNameList]
# Check if my signal name exists
if { [lsearch $signalList "top.dut0.din"] == -1 } {
  puts "signal top.dut0.din doesn't exist"
  ZEBU_exit
}
# Note: top.dut0.din is 32 bits length signal
# write signal binary value (completed with ‘0’ on MSB)
ZEBU_Signal_write top.dut0.din 1000011 %b
ZEBU_Monitor_flush
# read signal binary value
ZEBU_Monitor_refresh
set val [ZEBU_Signal_read top.dut0.din %b]
if {$val != "00000000000000000000000001000011"} {
  puts "Write/write binary failed"
  puts "binary value is $val"
}
# write signal octal value (completed with ‘0’ on MSB)
ZEBU_Signal_write top.dut0.din 7523 %o
ZEBU_Monitor_flush
# read signal octal value
ZEBU_Monitor_refresh
if { [ZEBU_Signal_read top.dut0.din %o] != "00000007523" } {
  puts "Write/write octal failed"
  puts "[ZEBU_Signal_read top.dut0.din %o]"
}
# write signal hexadecimal value (completed with ‘0’ on MSB)
ZEBU_Signal_write top.dut0.din BABE %h
ZEBU_Monitor_flush
# read signal hexadecimal value
ZEBU_Monitor_refresh
if { [ZEBU_Signal_read top.dut0.din %h] != "0000babe" } {
  puts "Write/write hexadecimal failed"
  puts "[ZEBU_Signal_read top.dut0.din %h]"
}
# write signal decimal value
ZEBU_Signal_write top.dut0.din 9023 %d
ZEBU_Monitor_flush
# read signal decimal value
ZEBU_Monitor_refresh
if { [ZEBU_Signal_read top.dut0.din %d] != "9023" } {
  puts "Write/write decimal failed"
  puts "[ZEBU_Signal_read top.dut0.din %d]"
}
```

4.4.7 Updating static probes

`ZEBU_CosimBuffer_IsUpdated` checks if the co-simulation buffer (shared between **zRun** and the testbench attached to **zRun**) is updated.

`ZEBU_CosimBuffer_Update` updates the co-simulation buffer (shared between **zRun** and the testbench attached to **zRun**)on the **zRun** side. This function updates the static probes values in **zRun**.

4.4.8 Deselecting signals from a Tcl script

`ZEBU_Signal_deselectAllSignals` allows you to deselect all the signals from a **zRun** Tcl script, including the ones previously selected via **zSelectProbes** or **zDbPostProc**.

4.5 Selecting sampling clocks for different features

This section is an important prerequisite to understanding the following sections:

- Section 4.6: Using the SRAM_TRACE feature
- Section 4.7: Using triggers with the Logic Analyzer
- Section 4.13: Advanced debug with Flexible Probes

`ZEBU_Clock_getSampleGroupNameList` returns a list of relevant clock group names (clock groups that contain available and valid sampling clocks).

`ZEBU_Clock_getSampleNameList` returns a list of available and valid (filtered) sampling clocks for trace, logical analysis, and Flexible Probes in a clock group.

Syntax:

```
ZEBU_Clock_getSampleGroupNameList <feature> <la_type>
ZEBU_Clock_getSampleNameList <clk_group> <feature> <la_type>
```

Where: `<feature>` is "`trace`", "`la`" (Logic Analyzer), or "`flp`" (Flexibles Probes)

`<la_type>` is "`TOT`" (Trace On Trig), or "`SOT`" (Stop On Trig), or "`none`".
Choose "`TOT`" or "`SOT`" if `<feature>` is "`la`"
Choose "`none`" if `<feature>` is NOT "`la`"

`<clk_group>` is one of the clock group names returned by `ZEBU_Clock_getSampleGroupNameList`

Some sampling clocks, as well as the clock group they belong to, may be filtered (for example `driverClk`, in-situ clocks, and free-running clocks).

Special Case: If you select a sampling clock at compilation (through a DVE file):

`ZEBU_Clock_getSampleGroupNameList "trace" "none"`	returns "`Hard wired`"
`ZEBU_Clock_getSampleNameList 0 "trace" "none"`	returns sampling clock name

4.6 Using the SRAM_TRACE feature

Note: This function is not supported if your ZeBu system is a Software Development Platform (SDP) and it is only available if you have instantiated SRAM_TRACE in the DVE file. Use ZEBU_Trace_isDefined to check if you have access to this functionality.

If SRAM_TRACE has been instantiated in the DVE file as described in the ***ZeBu Compilation Manual***, you can use the Tcl commands listed in the following sections.

4.6.1 Initializing SRAM_TRACE

ZEBU_Trace_isDefined must be used before initializing SRAM_TRACE. This command checks if the design compilation includes the driver and if the functionality is available with your ZeBu system.

ZEBU_Trace_setPreTrigRatio is then used to initialize SRAM_TRACE and define the data trace memory storage length in percentage (0 to 100). This command is dynamic (available while emulation is running).

Syntax:

```
ZEBU_Trace_isDefined
ZEBU_Trace_setPreTrigRatio <ratio>
```

Where: <ratio> is the data trace memory storage length in percentage (0 to 100)

ZEBU_Trace_isDefined returns:

- 1 if the **Trace** functionality is available
- 0 otherwise

ZEBU_Trace_setPreTrigRatio returns the associated number of samples that will be stored in the pre-trigger trace memory (depending on the number of signals linked with SRAM_TRACE in the DVE file).

Example: Initialize SRAM_TRACE:

```
# is Trace available ?
if { [ZEBU_Trace_isDefined] == 0 } {
  puts "SRAM trace not available"
  ZEBU_exit
}
# set SRAM trace length to 90% and display the equivalent number of
samples
puts "[ZEBU_Trace_setPreTrigRatio 90] samples will be stored in the
allocated trace memory"
```

4.6.2 Starting, stopping, and dumping the trace

ZEBU_Trace_start command starts the capture in the trace memory. During the run, each defined edge of the associated trace clock stores a new sample in the trace memory. This command is dynamic (available while emulation is running).

`ZEBU_Trace_stop` stops the capture in the trace memory. This command is dynamic (available while emulation is running).

`ZEBU_Trace_dumpFile` stores the trace memory contents to a waveform file in `VCD`, `VPD`, `BIN` or `FSDB` format (see Note, end of Section 3.3.3). This command is dynamic (available while emulation is running), but it is only available after `ZEBU_Trace_stop` has been performed.

Syntax:

```
ZEBU_Trace_start <clk_name> <clk_edge>
ZEBU_Trace_stop
ZEBU_Trace_dumpFile    <file_name>
```

Where: `<clk_name>` is the clock name returned by `ZEBU_Clock_getSampleNameList`

`<clk_edge>` is the `<clk_name>` edge sampling. Must be set to `posedge`, `negedge` or `bothedge`.

`<file_name>` is the waveform dump filename (`.vcd`, `.vpd`, `.bin` or `.fsdb`), which can be read with an appropriate waveform viewer.

See Section 4.4.8 for details on the `ZEBU_Clock_getSampleNameList` command.

Example: Typical use of SRAM_TRACE:

```
# Get the first clock of the first clock group available for triggering
set groupList [ ZEBU_Clock_getSampleGroupNameList "trace" "none" ]
set groupName [ lindex $groupList 0 ]
set clkList   [ ZEBU_Clock_getSampleNameList $groupName "trace" "none" ]
set clkName   [ lindex $clkList 0 ]
# Init of SRAM_TRACE, set the ratio
# and display the associated number of samples
puts "[ ZEBU_Trace_setPreTrigRatio 50 ] samples to store."
# enable SRAM_TRACE
ZEBU_Trace_start clkName posedge
# run 10000 on the same clock to store 10000 samples
ZEBU_Clock_enable clkName 10000
while { [ ZEBU_Clock_getStatus clkName ] == "running" } { after 100 }
# disable SRAM_TRACE
ZEBU_Trace_stop
# store trace in VCD file
ZEBU_Trace_dumpFile trace.vcd
```

4.6.3 Getting information from SRAM_TRACE

You have access to several other **zRun** Tcl commands to get the Trace status while the emulation is running. Some of these commands are useful if the Trace feature is linked with the Logic Analyzer, as described in Section 4.7.6.

`ZEBU_Trace_getPreTrigSample`: If `SRAM_TRACE` linked with Logic Analyzer, returns the number of samples which can be stored in **pre-trigger** trace memory.

`ZEBU_Trace_getPostTrigSample`: If `SRAM_TRACE` linked with Logic Analyzer, returns the number of samples which can be stored in **post-trigger** trace memory.

`ZEBU_Trace_getPostTrigAddr`: If `SRAM_TRACE` linked with Logic Analyzer, returns the start address of **post-trigger** trace memory.

`ZEBU_Trace_getCurrentAddr`: Gets current address of `SRAM_TRACE`.

`ZEBU_Trace_isEmpty`: Returns `0` if at least 1 sample was stored in trace memory. Returns `1` otherwise.

`ZEBU_Trace_isFull`: Returns `1` if the allocated trace memory is 100% full.

`ZEBU_Trace_getSize`: Returns the trace memory size.

`ZEBU_Trace_hasLooped`: Returns `1` if the pre-trigger circular buffer is full.

4.7 Using triggers with the Logic Analyzer

You can define up to 16 triggers on signals in the DVE file (whatever their types, without any specific limitation for each type). Refer to the ***ZeBu Compilation Manual*** and the ***ZeBu Reference Manual*** for more information about triggers declaration.
The triggers declared in the DVE file can be used and/or controlled with Tcl commands. Two types of triggers are available: dynamic triggers (their expression can be reprogrammed during emulation) and static triggers.

4.7.1 Getting a trigger list

This command returns a list of trigger names declared in the DVE.

Syntax:

```
ZEBU_Trigger_getNameList
```

Example: Display a list of all the triggers declared in the DVE file.

```
# get trigger list
set trigList [ZEBU_Trigger_getNameList]
# display it
foreach trigger $trigList {puts "Trigger name: $trigger"}
```

4.7.2 Getting a trigger type

Determines whether a trigger has been declared as static or dynamic in the DVE file.

Syntax:

```
ZEBU_Trigger_isDynamic <trig_name>
```

Where: `<trig_name>` is a trigger name returned by `ZEBU_Trigger_getNameList`. This command returns `1` when the trigger type is dynamic, `0` otherwise.

Example: Display all types of triggers declared in the DVE file.

```
# get trigger list
set trigList [ZEBU_Trigger_getNameList]
# display every trigger name and type
foreach trigger $trigList {
  puts "Trigger name: $trigger"
  if { [ZEBU_Trigger_isDynamic $trigger] == "1" } {
    puts "$trigger is Dynamic"
  } else {
    puts "$trigger is Static"
  }
}
```

4.7.3 Setting a dynamic trigger expression

`ZEBU_Trigger_setExpr` sets a dynamic trigger type expression. This command returns `1` if the expression is not valid, `0` otherwise. Note that **o**nly dynamic triggers can be reprogrammed during an emulation session. Use `ZEBU_Trigger_isDynamic` to determine if a trigger is programmable.

Syntax:

```
ZEBU_Trigger_setExpr <trig_name> <trig_expr>
```

Where: `<trig_name>` is a trigger name returned by `ZEBU_Trigger_getNameList`

`<trig_expr>` is the new dynamic trigger expression; the syntax of the expression is the Verilog syntax restricted to "`signal==constant`". Note that `signal` must be affected to the dynamic trigger in DVE file. `signal` may also be a vector or a sub-vector and `constant` may accept logical operation `&&`. You may also force the trigger value to `1` or `0` (`signal=1` or `signal=0`).

Example:

DVE file compatible with the following Tcl example:

```
# Dynamic trig on top.rst input signal
# Note that .output_bin is used even the signal is an input
zceiTrigger dTrig_rst (.output_bin (rst));
# Dynamic trig on top.dut.d[4:2] buried sub-vector
# Note that you need to specify buried signal/vector as
# ZeBu static probe with zNetgen to assign it to a dynamic trigger
zceiTrigger dTrig_d_2to4 (.output_bin ({top_dut_d[4:2]}));
# Dynamic trig on top.dout[4:0] output vector
zceiTrigger dTrig_dout_0to4 (.output_bin ({dout[4:0]}));
```

Set dynamic triggers expression with signal, vector and sub-vector assigned to the triggers in the DVE file.

```
# dTrig_rst is assigned to input signal top.rst in the dve file
if { [ZEBU_Trigger_isDynamic "dTrig_rst"] != "1" } {
  puts "ERROR dTrig_rst: not a dynamic trigger"
} else {
  set ret [ZEBU_Trigger_setExpr "dTrig_rst" "rst==1"]
  if { $ret } { puts "ERROR dTrig_rst: Invalid expression" }
}
# dTrig_d_2to4 is assigned to buried sub-vector top.dut.d[2:4]
# in the dve file
if { [ZEBU_Trigger_isDynamic "dTrig_d_2to4"] != "1" } {
  puts "ERROR dTrig_d_2to4: not a dynamic trigger"
} else {
  set ret [ZEBU_Trigger_setExpr "dTrig_d_2to4" \
                                "top_dut_d\[2:4\]==3`h5"]
  if { $ret } { puts "ERROR dTrig_d_2to4: Invalid expression" }
}
# dTrig_dout_0to4 is assigned to output vector top.dout[0:4]
# in the dve file
if { [ZEBU_Trigger_isDynamic "dTrig_dout_0to4"] != "1" } {
  puts "ERROR dTrig_dout_0to4: not a dynamic trigger"
} else {
  # Note: for this example only sub-vector of top.dout[0:4] is used
  set ret [ZEBU_Trigger_setExpr "dTrig_dout_0to4" \
                                "dout\[1:2\]==3 && dout\[4\]==0"]
  if { $ret } { puts "ERROR dTrig_dout_0to4: Invalid expression" }
}
```

4.7.4 Getting a Dynamic Trigger Expression

`ZEBU_Trigger_getExpr` returns the expression of a dynamic trigger declared in the DVE file.

Syntax:

```
ZEBU_Trigger_getExpr <trig_name>
```

Where: `<trig_name>` is a trigger name returned by `ZEBU_Trigger_getNameList`

Example: Display current expression for all dynamic triggers.

```
# get trigger list
set trigList [ZEBU_Trigger_getNameList]
# display every trigger name, type and
# expression if trigger type is dynamic
foreach trigger $trigList {
  puts "Trigger name: $trigger"
  if { [ZEBU_Trigger_isDynamic $trigger] == "1" } {
    puts "$trigger is Dynamic"
    puts "$trigger expression : [ZEBU_Trigger_getExpr $trigger]"
  } else {
    puts "$trigger is Static"
  }
}
```

4.7.5 Using a Stop-On-Trig Logic Analyzer

`ZEBU_LA_sot` associates a trigger to the Stop-On-Trig Logic Analyzer. This analyzer stops all clocks on the trigger when active. This command is dynamic (available while emulation is running).

`ZEBU_LA_start` enables the Logic Analyzer. This command is dynamic (available while emulation is running).

`ZEBU_LA_stop` disables the Logic Analyzer. This command is dynamic (available while emulation is running).

Syntax:

```
ZEBU_LA_sot <trig_name>
ZEBU_LA_start <clk_name> <clk_edge>
ZEBU_LA_stop
```

Where:
- `<trig_name>` is a trigger name returned by `ZEBU_Trigger_getNameList`
- `<clk_name>` is a clock name returned by `ZEBU_Clock_getSampleNameList`
- `<clk_edge>` is a `<clk_name>` edge sampling (must be set to `posedge` or `negedge`)

Example:

Typical use of Stop-On-Trig Logic Analyzer:

```
# Get the first clock of the first clock group available for triggering
set groupList [ ZEBU_Clock_getSampleGroupNameList "la" "SOT" ]
set groupName [ lindex $groupList 0 ]
set clkList   [ ZEBU_Clock_getSampleNameList $groupName "la" "SOT" ]
set clkName   [ lindex $clkList 0 ]
# Associate trigger tg0 to the SOT Logic Analyzer
ZEBU_LA_sot tg0
# Enable it
```

```
ZEBU_LA_start $clkName posedge
# Start Free running emulation
ZEBU_Clock_enableForever $clkName
# Wait until zebu Logic Analyzer stops all clock
while { [ZEBU_Clock_getStatus $clkName] == "running" } {}
# Disable zebu Logic Analyzer
ZEBU_LA_stop
# Get the cycle number executed before posedge trigger occurred
puts "Emulation stopped on trigger tg0 at cycle:"
puts "[ZEBU_Clock_getCounter $clkName]"
```

4.7.6 Using a Trace-On-Trig Logic Analyzer

Note: This feature is not supported if your ZeBu system is a Software Development Platform (SDP) and it is only available if you have instantiated SRAM_TRACE in the DVE file. Use ZEBU_Trace_isDefined to check if you have access to this functionality.

It is possible to declare signals in SRAM_TRACE and use the Logic Analyzer to drive the trace process. As long as the selected trigger does not occur, data are stored in the pre-trigger trace memory. Once the trigger occurred, data are stored in the post-trigger trace memory.

ZEBU_Trace_setPreTrigRatio initializes SRAM_TRACE and defines the ratio in percentage (0 to 100) between pre-trigger and post-trigger trace memory. This command is dynamic (available while emulation is running).

ZEBU_Trace_start starts the capture in the trace memory. During the run, each defined edge cycle of the associated trace clock stores a new sample in the trace memory. This command is dynamic (available while emulation is running).

ZEBU_LA_tot associates a trigger to the Trace-On-Trigger Logic Analyzer. This command is dynamic (available while emulation is running).

ZEBU_LA_start enables the Logic Analyzer. This command is dynamic (available while emulation is running).

ZEBU_Trace_stop stops the capture in the trace memory. This command is dynamic (available while emulation is running).

ZEBU_LA_stop disables the Logic Analyzer. This command is dynamic (available while emulation is running).

ZEBU_Trace_dumpFile stores the trace memory contents to a waveform file in VCD, VPD, BIN or FSDB format (see Note, end of Section 3.3.3). This command is dynamic (available while emulation is running), but it is only available after ZEBU_Trace_stop has been executed.

Syntax:

```
ZEBU_Trace_setPreTrigRatio <ratio>
ZEBU_Trace_start <clk_name> <clk_edge>
ZEBU_LA_tot <trig_name>
ZEBU_LA_start <clk_name> <clk_edge>
ZEBU_Trace_stop
ZEBU_LA_stop
ZEBU_Trace_dumpFile <file_name>
```

Where: `<ratio>` is the ratio (0 to 100%) between pre-trigger and post-trigger trace memory
`<trig_name>` is a trigger name returned by `ZEBU_Trigger_getNameList`
`<clk_name>` is a clock name returned by `ZEBU_Clock_getSampleNameList`
`<clk_edge>` is a `<clk_name>` edge sampling; must be set to `posedge` or `negedge`
`<file_name>` is the waveform dump filename (`.vcd`, `.vpd`, `.bin` or `.fsdb`), which can be read with an appropriate waveform viewer.

Notes:

- After disabling the *Trace on Trig* feature, you need to completely re-initialize the SRAM_TRACE. Use all these functions again if you want to use Trace with the Logic Analyzer in the same session.
- You have access to several other **zRun** Tcl commands to get the Trace status during the run, as described in Section 4.6.3.

Example: Typical use of Trace-On-Trig Logic Analyzer

```
# Get the first clock of the first clock group available for triggering
set groupList [ ZEBU_Clock_getSampleGroupNameList "la" "TOT" ]
set groupName [ lindex $groupList 0 ]
set clkList   [ ZEBU_LA_getSampleNameList $groupName "la" "TOT" ]
set clkName   [ lindex $clkList 0 ]
# Init of SRAM_TRACE, set the ratio
# and display the associated number of samples
puts "[ ZEBU_Trace_setPreTrigRatio 50 ] pre-trigger samples to store."
# enable SRAM_TRACE
ZEBU_Trace_start clkName posedge
# Associate trigger tg0 to TOT Logic analyzer
ZEBU_LA_tot tg0
# Enable TOT logic analyzer with the same clock as SRAM_TRACE
ZEBU_LA_start clkName posedge
# run 10000
# Note: in this example we suppose that tg0 occurs during this run
ZEBU_Clock_enable clkName 10000
while { [ ZEBU_Clock_getStatus clkName ] == "running" } { after 100 }
# disable SRAM_TRACE
ZEBU_Trace_stop
# stop TOT logic analyzer
ZEBU_LA_stop
# store trace in VCD file
ZEBU_Trace_dumpFile trace.vcd
```

4.7.7 Using the Flp On Trig feature

There is no **zRun** Tcl command integrating the **Flp on Trig** feature, but a script can be created with several commands for an equivalent scenario. See Section 3.5.3 for details on GUI usage of the **Flp on Trig** feature.

Example: Typical use of the **Flp on Trig** feature.
To run 2,000 cycles and activate the `shift` trigger afert 100 cyles:

```
# init FLP
ZEBU_Flp_isDefined
ZEBU_Flp_init
# Program SOT
ZEBU_LA_sot shift
```

```
ZEBU_LA_start clk posedge
# Start run 2000 (for Free run use ZEBU_Clock_enableForever)
ZEBU_Clock_enable clk 2000
while { [ZEBU_Clock_getStatus clk]=="running" } { after 100 }

# --- trigger fired ----

# invalidate SOT
ZEBU_LA_stop
# Program FLP
ZEBU_Flp_setSamplingClock "posedge clk"
ZEBU_Flp_addGroup ZLT_DFLT_GRP
ZEBU_Flp_setOutputDir ./flp.ztdb
ZEBU_Flp_probeOnOff "on"
# Restart clock (for Free run use ZEBU_Clock_enableForever and do not use
ZEBU_LA_clockRestart)
# THIS IS A NEW zRun FUNCTION to use only in Run mode
ZEBU_LA_clockRestart clk
while { [ZEBU_Clock_getStatus clk]=="running" } { after 100 }

# --- run 2000 finished ----

# dump FLP
ZEBU_Flp_probeOnOff "off"
ZEBU_Flp_close
ZEBU_Flp_flush
ZEBU_exit
```

4.8 Generating waveform dump files

During emulation, you can generate waveform files in `VCD`, `VPD`, `BIN` or `FSDB` format (see Note, end of Section 3.3.3) with the selected dynamic probes. You have access to two sampling modes:

- **Asynchronous:** Only if `-debugDriverClk` is used in the **`zRun`** command line. The fastest clock (`driverClk`) becomes the signals sampling clock.
- **Synchronous:** One of the design clocks is used as the signals sampling clock.

4.8.1 Monitoring a dump control

`ZEBU_Dump_file`: Before any waveform dump is performed, you need to define a filename and the sampling clock via the `ZEBU_Dump_file` command. This definition can be accomplished even if the dump file is not driven, thus it will have no impact on the run speed. The name of the file must end with `.vcd` for a `VCD` file, `.vpd` for a `VPD` file, `.bin` for a binary file, or `.fsdb` for an `FSDB` file (see Note, end of Section 3.3.3).

`ZEBU_Dump_flush`: Flushes the waveform file opened with `ZEBU_Dump_file` in order to read its contents before closing it with `ZEBU_Dump_close`.

`ZEBU_Dump_on` enables the waveform dump. This command is not dynamic, therefore you must use it when all clocks are stopped. You must use this command after a `ZEBU_Dump_file` command is sent, which creates the dump file and associates the sampling clock to the dump process. Several runs can be done between `ZEBU_Dump_file` and `ZEBU_Dump_on`.

`ZEBU_Dump_close` must be sent before any new dump file is specified with `ZEBU_Dump_file`.

`ZEBU_Dump_off` disables the waveform dump. This command is not dynamic, therefore you must use it when all clocks are stopped.

Syntax:

```
ZEBU_Dump_file <file_name> <sampling_clk>[signalListFilename=""
[compressionLevel=0 [dumpVirtualTime=0 [timeScale=-9 (for 1ns)]]]]
ZEBU_Dump_flush <file_name>
ZEBU_Dump_on
ZEBU_Dump_off
ZEBU_Dump_close
```

Where:	`<file_name>`	is the dump filename, which can be read with a appropriate waveform viewer (extension is `.vcd`, `.vpd`, `.bin` or `.fsdb`)
	`<sampling_clk>`	is a clock name returned by `ZEBU_Clock_getNameList`
	`[signalListFilename]`	forces internal nets while emulation is running, by adding all selected signals in the waveform file. This argument is the name of a file which lists the signals to be dumped. In this file, each line contains the hierarchical name of a signal and ends with an EOL character (a line which does not contain any space character).
	`[compressionLevel]`	modifies the actual compression for VCD and binary waveform files. The value is an integer `[0..9]`, `9` is the strongest compression. By default, `compressionLevel`=0.
	`[dumpVirtualTime]`	is intended to change the clock on which the sampling data are dated in the waveform file, using a virtual clock declared in the designFeatures file. It makes it possible to get waveform files with a timescale independent of the sampling method, thus allowing merge and comparison of waveforms created from different features (dynamic probes, flexible probes, HDL simulation, etc). When set to `1`, virtual time dumping is active. Default value is `0`, so that the waveforms are dated with a number of cycles of sampling clock.
	`[timeScale]`	is a power of ten (for example, `2` is for a 100-second timescale, `0` is for a 1-second timescale. By default timescale=-9 for 1 ns.

Notes:

- To use the **Asynchronous mode** use `driverClk` as `<sampling_clk>` (only available if `-debugDriverClk` is used in the **`zRun`** command line). Remember that when you use `driverClk`, no sampling clock starts until `driverClk` is started.
- After `ZEBU_Dump_off` command you can use `ZEBU_Dump_on` to continue the waveform dump in the same file, but it is NOT possible to define several dump file in the same emulation session.
- When dumping into an FSDB waveform file, the file writing process can be accelerated by setting the `ZEBU_FSDB_DUMP_USE_THREAD` environment variable to `YES`. This parallelization should be enabled only if there are many events per sampling cycle in the waveform file. Activating simultaneously parallelization for dumping and parallelization for file writing does not provide improvement of the performance but the memory and CPU footprints are higher.

Example:
Waveform dump in synchronous mode.

```
# create FDSB file named "readback.fsdb"
# trace all the signals whose names are in "../src/trace_list.txt"
# do not compress the wavefile
# enable virtual time conversion with a time scale of 10^-9s (1ns)
ZEBU_Dump_file readback.fsdb clk_c "../src/trace_list.txt" 0 1 -9
# Run 1000 on design clock clk1
ZEBU_Clock_enable clk1 1000
while { [ZEBU_Clock_getStatus clk1] == "running" } { after 500 }
# enable Waveform dump for 1000 clk1 cycles
ZEBU_Dump_on
ZEBU_Clock_enable clk1 1000
while { [ZEBU_Clock_getStatus clk1] == "running" } { after 100 }
# disable Waveform dump for 1000 clk1 cycles
ZEBU_Dump_off
ZEBU_Clock_enable clk1 1000
while { [ZEBU_Clock_getStatus clk1] == "running" } { after 100 }
# enable Waveform dump for 500 clk1 cycles
ZEBU_Dump_on
ZEBU_Clock_enable clk1 500
while { [ZEBU_Clock_getStatus clk1] == "running" } { after 100 }
# disable Waveform dump and exit
ZEBU_Dump_off
ZEBU_Dump_close
ZEBU_exit
```

4.8.2 Monitoring a dump status

`ZEBU_Dump_getStatus` returns the waveform dump status.

Syntax:

```
ZEBU_Dump_getStatus
```

Example: Display the waveform dump status:

```
if { [ZEBU_Dump_getStatus]=="dump_on" } {
  puts "Internal signal capture is turned on"
} else {
  puts "Internal signal capture is turned off"
}
```

4.8.3 Monitoring a dump file status

`ZEBU_Dump_getFileStatus` returns information on current file status, especially if the file is being written.

Syntax:

```
ZEBU_Dump_getFileStatus
```

Returns `file_opened` or `file_closed`.

4.9 Save & Restore operations

4.9.1 Saving/Restoring a logic state

`ZEBU_saveLogicState` saves the logic state of a ZeBu session into a form that is independent of the hardware platform type/configuration. The state of the session can then be restored on any hardware platform type/configuration using `ZEBU_restoreLogicState`.

`ZEBU_restoreLogicState` restores the logic state of a ZeBu session on any type of platform and in any configuration.

Syntax:

```
ZEBU_saveLogicState <LogicStateFile>
ZEBU_restoreLogicState <LogicStateFile>
```

Where: `<LogicStateFile>` is the name of the logic state you want to save/restore

4.9.2 Saving/Restoring a hardware state

`ZEBU_saveHardwareState` saves the hardware state of a ZeBu session. Also allows quick restore of the session status on the same hardware platform, using `ZEBU_restoreHardwareState`.

`ZEBU_restoreHardwareState` quickly restores the hardware state of a ZeBu session on a hardware platform. The state of design must have been saved as a hardware state (using `ZeBu_saveHardwareState`) and must be restored on the same hardware platform type/configuration as that on which the state of design had been saved.

Syntax:

```
ZEBU_saveHardwareState <HardwareStateFile>
ZEBU_restoreHardwareState <HardwareStateFile>
```

Where: `<HardwareStateFile>` is the name of the hardware state you want to save/restore

Note: You can use the `libZebuRestore` library to convert into a logic state any hardware state previously saved on any hardware platform type/configuration. You can then load this logic state onto any hardware platform type/configuration.

Example:

```
ZEBU_open
ZEBU_saveHardwareState zebu.hardware.state
ZEBU_close
ZEBU_restoreHardwareState zebu.hardware.state
```

4.10 Controlling dynamic forces

`ZEBU_Signal_force` forces the value of a signal.

`ZEBU_Signal_release` unassigns a signal which has been assigned by a previous call of `ZEBU_Signal_force`.

`ZEBU_Signal_isForceable` test if a signal can be forced or not.

Syntax:

```
ZEBU_Signal_force <signalName> <value> [format>
ZEBU_Signal_release <signalName>
ZEBU_Signal_isForceable <signalName>
```

To improve the runtime emulation frequency, there is no automatic writing into the hardware when a dynamic force is controlled from the user application. The actual writing must be initiated manually using the `ZEBU_Monitor_flush` command. The change is immediate and does not require to step in the driver clock.

Note: The performance improvement is mostly sensible when many dynamic forces are modified simultaneously because the actual writing is done only once.

4.11 Controlling DPI function calls

When there is no C/C++ testbench in the environment, the control of the DPI function calls can be done directly from **zRun**, using the following commands:

- `ZEBU_CCall_selectSamplingClockGroup clockGroupName`
- `ZEBU_CCall_selectSamplingClocks clockExpression`
- `ZEBU_CCall_enableSynchronization`
- `ZEBU_CCall_disableSynchronization`
- `ZEBU_CCall_setOnEvent`
- `ZEBU_CCall_unsetOnEvent`
- `ZEBU_CCall_loadDynamicLibrary fullname`
- `ZEBU_CCall_start [scope [importName [callNumber]]]`
- `ZEBU_CCall_stop [scope [importName [callNumber]]]`
- `ZEBU_CCall_start2 scopeExpression [invert=0 [ignoreCase=0 [hierarchicalSeparator=. [importName [callNumber]]]]]`
- `ZEBU_CCall_stop2 scopeExpression [invert=0 [ignoreCase=0 [hierarchicalSeparator=. [importName [callNumber]]]]]`

For functional details and for information about parameters, you should refer to the equivalent method of the C++ API described in *Application Note AN029*.

When the DPI calls implemented in the dynamic library (`dpi.so`) are controlled from **zRun**, the testbench only controls the co-simulation driver and a specific script is written for **zRun** (`script_dpi.tcl`). DPI calls cannot be controlled simultaneously from two processes.

4.11.1 Example

Example of `script_dpi.tcl` script:

```
# open the board
ZEBU_open
# load of the DPI dynamic library
ZEBU_Call_loadDynamicLibrary dpi.so
# selection of both edges of clk to sample the DPI calls
ZEBU_CCall_selectSamplingClocks clk
# Start of all the DPI calls
ZEBU_CCall_start
# run the clocks clk
```

```
ZEBU_Clock_enable clk 200000
while { [ZEBU_getStatus] == "open" && [ZEBU_Clock_getStatus clk] ==
"running" } { after 500  }
# stop dpi processing
ZEBU_CCall_stop
# close the board
ZEBU_close
```

4.11.2 Proceeding with runtime

zRun is launched to run both the testbench and the Tcl script which controls the DPI calls, after the correct setting of LD_LIBRARY_PATH environment variable:

```
$ export LD_LIBRARY_PATH=<path to dpi.so>:$LD_LIBRARY_PATH
$ zRun -testbench ./tb -do script_dpi.tcl
```

4.12 Randomization operations

4.12.1 Selecting signals/memories for randomization

ZEBU_selectSignalsToRandomize is used to select a set of signals which will be randomized.

ZEBU_selectMemoriesToRandomize is used to select a set of memories which will be randomized.

Syntax:

```
ZEBU_selectSignalsToRandomize <signalList> <invert>
ZEBU_selectMemoriesToRandomize <memoryList> <invert>
```

Where:

<signalList>	is the name of a file containing a list of signals to randomize or not randomize. The file must contain a list of hierarchical names of signals separated by <EOL> characters.
<memoryList>	is the name of a file containing a list of memories to randomize or not randomize. The file must contain a list of hierarchical names of memories separated by <EOL> characters.
<invert>	is an integer (0 or 1) used to invert the meaning of signals/memories lists: 0: All signals/memories belonging to specified list will be randomized. 1: All signals/memories not belonging to specified list will be randomized.

4.12.2 Pseudo-randomizing signals /memories

ZEBU_randomizeSignals is used to set signals to pseudo-random values. The signals to be modified should have been previously selected by one or more calls to ZEBU_selectSignalsToRandomize.

ZEBU_randomizeMemories is used to set memories to pseudo-random values. The memories to modify should have been previously selected by one or more calls to ZEBU_selectMemoriesToRandomize.

ZEBU_randomize is used to set signals and memories to pseudo-random values. The signals and memories to modify should have been previously selected by one or more calls to ZEBU_selectSignalsToRandomize and ZEBU_selectMemoriesToRandomize.

Syntax:

```
ZEBU_randomizeSignals <seed>
ZEBU_randomizeMemories <seed>
ZEBU_randomize <seed>
```

Where: `<seed>` is the seed of the sequence of values you want to set as pseudo-random

4.13 Advanced debug with Flexible Probes

The group of Flexible Probes declared at compilation can be controlled by Tcl script.

4.13.1 Activating the Flexible Probes feature

`ZEBU_Flp_isDefined` returns `1` if Flexible Probes feature is available; else `0`.

Syntax:

```
ZEBU_Flp_isDefined
```

Note: No other Tcl command controlling the Flexible Probes will be valid until the `ZEBU_Flp_isDefined` command is invoked.

Example:

```
# is Flp feature available ?
if { [ZEBU_Flp_isDefined] == 0 } {
     puts "No Flexible Probes available"
     ZEBU_exit
}
```

4.13.2 Initializing a Flexible Probe

`ZEBU_Flp_init` initializes the Flexible Probe drivers and does not return anything.

Syntax:

```
ZEBU_Flp_init
```

You must invoke this command immediately after `ZEBU_Flp_isDefined` is sent.

Example:

```
# is Flp feature available ?
if { [ZEBU_Flp_isDefined] == 0 } {
     puts "No Flexible Locale Probes defined"
     ZEBU_exit
}
ZEBU_Flp_init
```

4.13.3 Setting the sampling clock for Flexible Probes

`ZEBU_Flp_setSamplingClock` sets the sampling clock for Flexible Probes.

Note: This command is only useful if the sampling clock and clock edge were not declared at compilation through the `probe_signals -clock_name` command (hard-wired mode). Refer to ***ZeBu-Server Compilation Manual*** for details on the `probe_signals` command.

Syntax:

```
ZEBU_Flp_setSamplingClock <clock_expression>
```

Where: `<clock_expression>` has the following format:
`[posedge|negedge] <clock name>[ or [posedge|negedge] <clock name>]*`

If this command is invoked in hard-wired mode, the clock setting is ignored, and the following message is displayed:

```
Warning: "Hard wired" Flex Probes mode detected. Your clock selection <clock> has not been
taken into account.
```

In the specific case of `driverClk` being used as a sampling clock, `posedge` is the only valid edge.

Example: To select a sampling clock and a clock edge:

```
proc setFlpClk { my_clk my_edge } {
    if { $my_clk == "driverClk"} {
        # Don't care about $my_edge, only "posedge" is available
        ZEBU_Flp_setSamplingClock "posedge driverClk"
    } else {
        # Check $my_edge is valid
        if { $my_edge != "posedge" && $my_edge != "negedge" } {
            puts "ERROR : Cannot set Flp sampling edge : $my_edge is not valid"
            return
        }
        # Check $my_clk is valid
        foreach domain [ZEBU_Clock_getSampleGroupNameList flp none] {
            foreach clk [ZEBU_Clock_getSampleNameList $domain flp none] {
                if { $my_clk == $clk } {
                    puts "Flp sampling clock is set to $my_clk with $my_edge"
                    ZEBU_Flp_setSamplingClock "$my_edge $my_clk"
                    return
                }
            }
        }
        puts "ERROR : Cannot set Flp sampling clock : $my_clk is not valid"
    }
}
```

4.13.4 Selecting a group

Use `ZEBU_Flp_getGroupNameList` to retrieve the list of Flexible Probes declared at compilation. Refer to the ***ZeBu-Server Compilation Manual*** for details on declaring Flexible Probes for compilation.

Use `ZEBU_Flp_addGroup` to make a group eligible for dumping.

Syntax:

```
ZEBU_Flp_getGroupNameList <stringFilter>
ZEBU_Flp_addGroup <groupName>
```

Where: `<stringFilter>` is an argument for group filtering (patterns matching). The filter is based on the `fnmatch()` function (from standard C library `<fnmatch.h>`) with flags argument positionned at `FNM_NOESCAPE`

`<groupName>` is one of the groups returned by `ZEBU_Flp_getGroupNameList`

Example: To validate all groups containing an `input` occurrence.

```
set groups [ZEBU_Flp_getGroupNameList "*input*"]
if { [llenght $groups] == 0 } {
     puts "ERROR : No group contains \"*input*\" occurence"
} else {
     foreach grp $groups { ZEBU_Flp_addGroup $grp }
}
```

4.13.5 Selecting an output directory for Flexible Probes

Use `ZEBU_Flp_setOutputDir` to specify a dump directory (raw format) for Flexible Probes.

Syntax:

```
ZEBU_Flp_setOutputDir <dirName> [compressionLevel=0 [dumpVirtualTime=0
[timeScale=-9 (for 1ns)]]]
```

Where:

`<dirName>`	is the Flexible Probes dump directory (raw `ztdb` format).
`[compressionLevel]`	modifies the actual compression for VCD and binary waveform files. The value is an integer `[0..9]`, where `9` is the strongest compression. By default, `compressionLevel`=0.
`[dumpVirtualTime]`	is intended to change the clock on which the sampling data are dated in the waveform file, using a virtual clock declared in the `designFeatures` file. It makes it possible to get waveform files with a timescale independent of the sampling method, thus allowing merge and comparison of waveforms created from different features (dynamic probes, flexible probes, HDL simulation, etc). When set to `1`, virtual time dumping is active. The default value is `0`, so that the waveforms are dated with a number of cycles of the sampling clock.
`[timeScale]`	is a power of ten (for example, `2` is for a 100-second timescale, `0` is for a 1-second timescale. By default timescale=-9 for 1 ns.

Example:

```
# Set the output dir to "flexible.fsdb"
# no compression
# enable virtual time conversion with a time scale of 1ns (10^-9s)
ZEBU_Flp_setOutputDir flexible.fsdb 0 1 -9
```

4.13.6 Activating Flexible Probes at runtime

Use `ZEBU_Flp_probeOnOff` to control Flexible Probes activation at runtime.

Syntax:

```
ZEBU_Flp_probeOnOff <mode>
```

Where:

`<mode>`	is one of two valid modes: `ON` or `OFF` (case insensitive)

Example: To run 100 cycles in ON mode, then 100 cycles in OFF mode, then 100 cycles in ON mode.

```
# Note : in this example "clk" is not necessarily the Flp Sampling Clock
# set Flp in probe ON mode
ZEBU_Flp_probeOnOff "on"
# run 100 cycles on clk
ZEBU_Clock_enable clk 100
while { [ZEBU_Clock_getCounter clk] != 100 } { after 100 }
# set Flp in probe OFF mode
ZEBU_Flp_probeOnOff "off"
```

```
# run 100 cycles on clk
ZEBU_Clock_enable clk 100
while { [ZEBU_Clock_getCounter clk] != 100 } { after 100 }
# set Flp in probe ON mode
ZEBU_Flp_probeOnOff "on"
# run 100 cycles on clk
ZEBU_Clock_enable clk 100
while { [ZEBU_Clock_getCounter clk] != 100 } { after 100 }
```

4.13.7 Flushing Flexible Probes

`ZEBU_Flp_flush` can be useful to view the entire content of the Flexible Probes without closing the current session.

`ZEBU_Flp_close` performs a flush before saving files in a raw format (see Section 4.13.5) and closing the session.

Syntax:

```
ZEBU_Flp_flush
ZEBU_Flp_close
```

Example:

To run 100 cycles in ON mode, then flush, then run 100 cycles in ON mode, then close the session.

```
# Note : in this exemple "clk" is not necessarily the Flp Sampling Clock
# set Flp in probe ON mode
ZEBU_Flp_probeOnOff "on"
# run 100 cycles on clk
ZEBU_Clock_enable clk 100
while { [ZEBU_Clock_getCounter clk] != 100 } { after 100 }
# Flush without closing Flp session
ZEBU_Flp_flush
# Put your own foreground procedure to convert raw data to waveforms (see 4.13.8)
# run 100 cycles on clk
ZEBU_Clock_enable clk 100
while { [ZEBU_Clock_getCounter clk] != 100 } { after 100 }

# close Flp session
ZEBU_Flp_close
```

4.13.8 Converting raw ZTDB to VCD or FSDB formats in Tcl script

Example:

```
proc checkFlpPostProc { nproc } {
      if { [catch {exec ps $nproc}] != 1 } { after 100 checkFlpPostProc $nproc }
}
# Only available if you are connected to the ZeBu system
set zebuWork [ZEBU_getZebuWork]
set flpDir "./flp.ztdb"
set outputFileVcd   "./flp.vcd"
set outputFileFsdb  "./flp.fsdb"
# Background vcd file generation
checkFlpPostProc [eval exec ztdb2vcd -i $flpDir -o $outputFileVcd -z $zebuWork &]
# Background fsdb file generation
checkFlpPostProc [eval exec ztdb2fsdb -i $flpDir -o $outputFileFsdb -z $zebuWork &]
# Foreground vcd file generation
eval exec ztdb2vcd -i $flpDir -o $outputFileVcd -z $zebuWork
# Foreground fsdb file generation
eval exec ztdb2fsdb -i $flpDir -o $outputFileFsdb -z $zebuWork
```

Note: If you need to convert raw `ZTDB` to `VCD` or `FSDB` formats independently of **`zRun`**, look for **`ztdb2vcd`** and **`ztdb2fsdb`** in the *ZeBu API Reference Manual*.

4.14 Additional Tcl commands

4.14.1 `ZEBU_getDoFile`

Use this command to get the filename given to **`zRun`** with the `-do` option. It takes no argument and returns the name of the script. This script can then be executed using the `source` command.

Example:

```
# get the -do argument and execute the returned script
if { [ZEBU_getDoFile] != "" } { source $doFile }
```

4.14.2 `ZEBU_getSeparator`

Use this command to find out what type of character is specified as a separator for signals and memories.

Syntax:

```
ZEBU_getSeparator
```

4.14.3 `ZEBU_sleep`

Use this command to specify the amount of time to wait.

Syntax:

```
ZEBU_sleep <ms>
```

Where: `<ms>` is the waiting period in seconds (1,000 ms)

4.14.4 `ZEBU_getZebuFamily`

Use this command to identify the ZeBu type detected at system connection. Returned information is: `ZeBu-ZSE`, `ZeBu-PE`, or `Unknown`.

4.14.5 `ZEBU_debugDriverClk`

Use this command to find out if **`zRun`** was launched with the `-debugDriverClk` Option. If it is the case `true` is returned; `false` otherwise.

4.14.6 `ZEBU_getDriverClkFrequency`

Use this command to get the frequency of `driverClk` (in kHz).

4.14.7 `ZEBU_Memory_erase`

Use this command to reset an entire memory to `0x0`.

Syntax:

```
ZEBU_Memory_erase <memoryName>
```

Where: `<memoryName>` is the name of the memory you want to erase

4.14.8 ZEBU_noClockRefresh

Use this command to find out if the `-noClockRefresh` option was used. If it is the case `true` is returned; `false` otherwise.

4.14.9 ZEBU_getZebuWork

Use this command to retrieve `zebu.work` for the current run. The connection to the ZeBu system must be established (via `ZEBU_open`).

4.14.10 ZEBU_Zemi_isDefined

Use this command to find out if one or more ZEMI-3 transactor modules were compiled with the design. Returns the number of transactor modules.

4.14.11 ZEBU_detach

When **zRun** is used with the `-testbench` option, use this command to close the **zRun** connection without disconnecting the testbench. This is particularly useful to stop overloading ZeBu with the **zRun** process as soon as **zRun** is disconnected.

4.14.12 ZEBU_synchroCloseStatus

When the `-synchroClose` option is used, the testbenches remain stuck to their call to the `Close` command while waiting for **zRun** to actually close the connection. This command thus allows you to find out if all testbenches are definitely disconnected. If it is the case `true` is returned; `false` otherwise.

4.14.13 ZEBU_getCoverage

Reads values from an input file (`emulator.probe_names`) and writes them into an output file (`emulator.values`).

4.14.14 ZEBU_isHDP

Detects at runtime if the system is HDP (Hardware Development Platform) before HDP-only features can be activated (Trace feature in particular).

4.15 Tcl commands list

The following table lists all the Tcl functions used with **zRun**. By default, all functions can be used in HDP and SDP except when one of these restrictions applies:

- A tick (✓) next to a function indicates that it can only be used in HDP.
- In SDP, some `ZEBU_memory_*` functions do not work on BRAMs. This is indicated by `BRAM+zrm` in the HDP column and `zrm` in the SDP column.
- In HDP, `ZEBU_randomizeMemories` works on BRAMs only.
- Use `ZEBU_isHDP` to detect if the system is HDP or SDP (see Section 4.14.14).

(*) HDP: Software development platform; SDP: Hardware development platform

Tcl Command	HDP (*)	SDP (*)	Section
ZEBU_Clock_disable			4.2.5
ZEBU_Clock_enable			4.2.3
ZEBU_Clock_enableForever			4.2.4
ZEBU_Clock_getCounter			4.2.7
ZEBU_Clock_getGroupNameList			4.2.1
ZEBU_Clock_getNameList			4.2.2
ZEBU_Clock_getSampleGroupNameList			4.4.8
ZEBU_Clock_getSampleNameList			4.4.8
ZEBU_Clock_getStatus			4.2.6
ZEBU_close			4.1.3
ZEBU_CosimBuffer_IsUpdated			4.4.7
ZEBU_CosimBuffer_Update			4.4.7
ZEBU_debugDriverClk			4.14.5
ZEBU_detach			4.14.11
ZEBU_Dump_close			4.8.1
ZEBU_Dump_file			4.8.1
ZEBU_Dump_flush			4.8.1
ZEBU_Dump_getFileStatus			4.8.3
ZEBU_Dump_getStatus			4.8.2
ZEBU_Dump_off			4.8.1
ZEBU_Dump_on			4.8.1
ZEBU_exit			4.1.5
ZEBU_Flp_addGroup	✓		4.13.4
ZEBU_Flp_close	✓		4.13.7
ZEBU_Flp_flush	✓		4.13.7
ZEBU_Flp_getGroupNameList	✓		4.13.4
ZEBU_Flp_init	✓		4.13.2
ZEBU_Flp_isDefined	✓		4.13.1
ZEBU_Flp_probeOnOff	✓		4.13.6
ZEBU_Flp_setOutputDir	✓		4.13.5
ZEBU_Flp_setSamplingClock	✓		4.13.3
ZEBU_getCoverage			4.14.13
ZEBU_getDoFile			4.14.1
ZEBU_getDriverClkFrequency			4.14.6
ZEBU_getSeparator			4.14.2
ZEBU_getStatus			4.1.4
ZEBU_getZebuFamily			4.14.4
ZEBU_getZebuWork			4.14.9
ZEBU_isHDP			4.14.14
ZEBU_LA_sot			4.7.5
ZEBU_LA_start			4.7.5, 4.7.6
ZEBU_LA_stop			4.7.5, 4.7.6
ZEBU_LA_tot	✓		4.7.6
ZEBU_Memory_depth			4.3.3
ZEBU_Memory_erase	BRAM+zrm	zrm	4.14.7
ZEBU_Memory_getNameList			4.3.1
ZEBU_Memory_isWritable			4.3.2
ZEBU_Memory_loadFromBuffer	BRAM+zrm	zrm	4.3.5
ZEBU_Memory_loadFromFile	BRAM+zrm	zrm	4.3.5
ZEBU_Memory_readWordBuffer			4.3.7

Tcl Command	HDP (*)	SDP (*)	Section
ZEBU_Memory_storeToBuffer			4.3.4
ZEBU_Memory_storeToFile			4.3.4
ZEBU_Memory_update			4.3.7
ZEBU_Memory_width			4.3.3
ZEBU_Memory_writeWordBuffer	BRAM+zrm	zrm	4.3.6
ZEBU_Monitor_flush	✓		4.4.6
ZEBU_Monitor_refresh			4.4.2
ZEBU_noClockRefresh			4.14.8
ZEBU_open			4.1.1
ZEBU_randomize			4.12.2
ZEBU_randomizeMemories	BRAM		4.12.2
ZEBU_randomizeSignals	✓		4.12.2
ZEBU_restoreHardwareState			4.9.2
ZEBU_restoreLogicState	✓		4.9.1
ZEBU_saveHardwareState			4.9.2
ZEBU_saveLogicState			4.9.1
ZEBU_selectMemoriesToRandomize	✓		4.12.1
ZEBU_selectSignalsToRandomize	✓		4.12.1
ZEBU_Signal_getLocalInstance			4.4.3
ZEBU_Signal_getLocalNode			4.4.2
ZEBU_Signal_getNameList			4.4.1
ZEBU_Signal_getType			4.4.2
ZEBU_Signal_getWidth			4.4.2
ZEBU_Signal_isForceable			4.10
ZEBU_Signal_isWritable			4.4.2
ZEBU_Signal_read			4.4.2
ZEBU_Signal_write	✓		4.4.6
ZEBU_Signal_deselectAllSignals			4.4.8
ZEBU_sleep			4.14.3
ZEBU_synchroCloseStatus			4.14.12
ZEBU_Trace_dumpFile	✓		4.6.2, 4.7.6
ZEBU_Trace_getCurrentAddr	✓		4.6.3
ZEBU_Trace_getPostTrigAddr	✓		4.6.3
ZEBU_Trace_getPostTrigSample	✓		4.6.3
ZEBU_Trace_getPreTrigSample	✓		4.6.3
ZEBU_Trace_getSize	✓		4.6.3
ZEBU_Trace_hasLooped	✓		4.6.3
ZEBU_Trace_isDefined	✓		4.6.1
ZEBU_Trace_isEmpty	✓		4.6.3
ZEBU_Trace_isFull	✓		4.6.3
ZEBU_Trace_setPreTrigRatio	✓		4.6.1, 4.6.2, 4.7.6
ZEBU_Trace_start	✓		4.6.2, 4.7.6
ZEBU_Trace_stop	✓		4.6.2, 4.7.6
ZEBU_Trigger_getExpr			4.7.4
ZEBU_Trigger_getNameList			4.7.1
ZEBU_Trigger_isDynamic			4.7.2
ZEBU_Trigger_setExpr			4.7.3
ZEBU_Zemi_isDefined			4.14.10

5 ZeBu Documentation Package

For each software version, the ***ZeBu Release Note*** describes the new features, the fixed bugs, the known limits, the evolutions of the documentation package and the compatibility information.

The following Manuals constitute the ZeBu documentation package (some of them are specific to one type of ZeBu system; others are generic manuals for the ZeBu range, as described in the ***ZeBu Release Note***):

[1] The ***ZeBu Installation Manual*** describes how to install the ZeBu software and hardware.

[2] The ***ZeBu Compilation Manual*** describes the ZeBu 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 Smart Z-ICE Manual*** provides detailed information on how to configure and to connect the Smart Z-ICE interface to an external system.

[11] The ***ZeBu Direct ICE Manual*** provides detailed information on how to configure and to connect an ICE module for connection emulated DUT I/O pins to a target system and hard cores.

[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.

6 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. #B-15, Raheja Arcade, 80 Ft. Road, 5th Block, Koramangala Bangalore - 560 095 Karnataka INDIA Tel: +91-80-41460680/30202343
Taiwan Headquarters	14F1, No 371, Sec. 1 Guangfu Rd., East District Hsinchu City 300 TAIWAN (R.O.C.) Tel: +886-(3)-564-7900