



The Fastest Verification

ZeBu™

Reference Manual

Document revision – b –

April 2011

Version 6_3_0

Copyright © 2002-2011 EVE. All rights reserved.

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



Copyright Notice Proprietary Information

Copyright © 2002-2011 EVE. All rights reserved.

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

Right to Copy Documentation

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

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

Destination Control Statement

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

Disclaimer

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



Table of Contents

ABOUT THIS MANUAL.....	6
OVERVIEW	6
HISTORY	6
RELATED DOCUMENTATION	6
1 INTRODUCTION	7
2 DVE FILE.....	8
2.1 INTRODUCTION	8
2.1.1 Description	8
2.1.2 Generating a DVE file template with <i>zCui</i>	9
2.2 SYNTAX FOR HIERARCHICAL PATHS IN THE DVE FILE.....	10
2.2.1 RTL hierarchical paths	10
2.2.2 Using identifiers.....	10
2.2.2.1 Simple identifiers	10
2.2.2.2 Hierarchical names	10
2.2.3 Using escaped characters in hierarchical names	11
2.3 DECLARING VERIFICATION DRIVERS.....	11
2.3.1 Instantiating the drivers	11
2.3.2 Declaring specific parameters for a driver	12
2.4 DECLARING CLOCK DRIVERS.....	14
2.4.1 Declaring primary controlled clocks	14
2.4.2 Declaring primary clocks coming from Direct ICE or Smart Z-ICE	14
2.4.3 Declaring frequency synthesizers for prototyping.....	14
2.5 DECLARING CLOCKS PARAMETERS FOR ZEBU SYSTEM CLOCKS.....	15
2.6 DECLARING PARAMETERS FOR ADVANCED CLOCK MODELING	15
2.6.1 Modifying the number of available controlled clocks.....	15
2.6.2 Clock generator accuracy	16
2.7 USING ASSIGN STATEMENTS.....	16
2.8 DECLARING REGISTERS	17
2.9 DECLARING TRIGGERS.....	17
2.9.1 Static triggers.....	17
2.9.2 Dynamic triggers	17
2.10 INSTANTIATING SIMPLE VERILOG LOGIC GATES.....	18
2.11 SAMPLING ON AN UNCONTROLLED CLOCK	19
2.12 GENERAL SYNTAX	20
2.13 EXAMPLES	23
2.13.1 DVE file instantiating an HDL_COSIM driver	23
2.13.2 DVE file instantiating a C_COSIM driver	24
2.13.3 DVE file instantiating an SRAM_TRACE driver.....	26



3	DESIGNFEATURES FILE.....	28
3.1	INTRODUCTION	28
3.1.1	<i>Syntax rules</i>	28
3.1.2	<i>designFeatures.<PCname>.help</i>	28
3.2	LOCATION OF THE CALIBRATION FILES	28
3.3	DECLARATIONS OF PROCESSES	29
3.3.1	<i>Single-process environment</i>	29
3.3.2	<i>Multi-process environment</i>	29
3.3.3	<i>Unidentified processes</i>	29
3.3.4	<i>When initialization phases not carried out on memory</i>	30
3.3.5	<i>zRun unidentified process</i>	30
3.4	DECLARATIONS FOR DESIGN CLOCKS.....	31
3.4.1	<i>Declaring Additional Clocks</i>	31
3.4.2	<i>Parameters for primary clocks</i>	31
3.4.2.1	<i>Syntax</i>	31
3.4.2.2	<i>Example of DUT clock description in controlled mode</i>	32
3.4.2.3	<i>Example of DUT clock description in free-running mode</i>	32
3.4.2.4	<i>Example of DUT clock description in in-situ mode</i>	32
3.4.2.5	<i>Optimizing the runtime performance with Tolerance</i>	33
3.4.2.7	<i>Clock declaration for HDL co-simulation</i>	34
3.4.2.8	<i>Clock declaration for C/C++ co-simulation</i>	34
3.4.3	<i>Declaration with a separate Clock File</i>	34
3.4.4	<i>Propagation delay</i>	34
3.4.5	<i>Offset delay for clock skews</i>	34
3.4.6	<i>Frequency synthesizers for prototyping</i>	35
3.5	DECLARATIONS FOR TRANSACTORS	36
3.5.1	<i>Global transactor settings</i>	36
3.5.2	<i>Specific transactor settings</i>	36
3.6	DECLARATION FOR MEMORY INITIALIZATION	36
3.7	DECLARATION OF SYSTEM CLOCK FREQUENCIES	37
3.8	DECLARATION FOR DRIVERCLK RESET SIGNAL.....	37
3.9	DECLARATIONS FOR DUT RESET	37
3.10	DECLARATION FOR DIRECT ICE.....	38
3.11	DECLARATION FOR SMART Z-ICE.....	38
3.12	DECLARATION FOR SVAS	38
4	RUNTIME CLOCK FILE.....	39
4.1	DESCRIPTION.....	39
4.2	CLOCK FILE CONTENT	39
4.3	GENERAL SYNTAX	39



5	MEMORY CONTENT FILE	40
5.1	INTRODUCTION	40
5.2	DESCRIPTION.....	40
5.3	EXAMPLE	41
5.4	MEMORY INITIALIZATION IN THE VERILOG SOURCE FILE.....	42
6	PARFF PARAMETER FILE	43
6.1	MODIFICATION OF THE PARFF PARAMETER FILE	43
6.1.1	<i>Modifying FPGA compilation parameters.....</i>	<i>44</i>
6.1.2	<i>Modifying PARFF configuration</i>	<i>45</i>
6.1.3	<i>Reserved parameters.....</i>	<i>45</i>
6.2	EXAMPLE: DEFAULT PARFF PARAMETER FILE.....	45
7	ZEBU-SERVER DOCUMENTS	47
8	EVE CONTACTS.....	48

Figures

Figure 1: ZeBu flow with input files for compilation and runtime.....	7
Figure 2: Instantiation of an HDL_COSIM driver in the DVE file	23
Figure 3: Instantiation of a C_COSIM driver in the DVE file.....	24
Figure 4: Instantiation of an SRAM_TRACE driver in the DVE file	26

Tables

Table 1: List of drivers provided in the ZeBu package	12
Table 2: Predefined parameters for transaction-based verification	13
Table 3: ZeBu System Clocks with frequency constraints for compilation.....	15
Table 4: Allowed system clock frequencies for runtime.....	37
Table 5: FPGA P&R parameter sets in the PARFF parameter file.....	43
Table 6: Options for ISE Place and Route	44
Table 7: Other Options for FPGA Compilation	44



About This Manual

Overview

This manual describes the input files for compilation and runtime operations which are specific to ZeBu environment. These file formats are described with their general syntax, detailed information about their elements, and examples.

History

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

Doc Revision	Product Version	Date	Evolution
b	6_3_0	Apr 11	Full revision for V6_3_0.
a	1.3_x	Feb 06	First Edition for ZeBu-XL only.

Related Documentation

The complete ZeBu documentation package is listed at the back of this manual, in Chapter 7.

1 Introduction

The following figure shows the input files created by user to compile and emulate a design with ZeBu. The ZeBu proprietary files described in this manual are visible in green. Information about other input files, which are design source files or scripts, is available in the *ZeBu-Server Compilation Manual*.

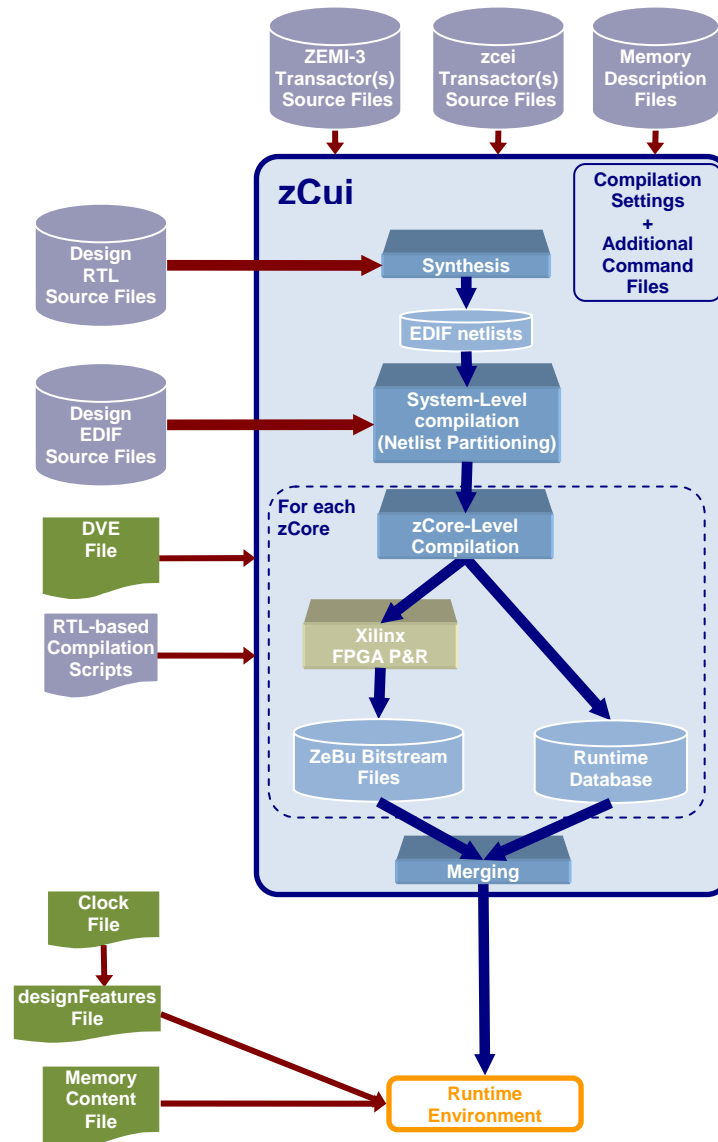


Figure 1: ZeBu flow with input files for compilation and runtime



2 DVE file

2.1 Introduction

A ZeBu proprietary file, known as the DVE file (Design Verification Environment), describes the connection between the design and its environment. This file is a mandatory input for the ZeBu compiler, used by for system-level compilation and for RTB compilation.

The DVE file is generally written manually. It requires a good knowledge of the design and the verification environment.

A DVE file template can be generated by the ZeBu compiler for event-based HDL co-simulation, cycle-based C/C++ co-simulation, or SRAM trace. It is easy to re-use a DVE file or pieces of a DVE file for multiple similar designs or interfaces.

The syntax for the DVE file is similar to Verilog syntax.

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

The ZeBu compiler checks the content of the DVE file and determines whether or not system-level compilation should be launched (incremental compilation). In case the latest DVE modifications impact only the RTB compiler (for example, when a transactor is added or removed from the environment), then **zCui** will skip the system-level compilation step.

2.1.1 Description

The DVE file describes the interconnection of your DUT with clock drivers and verification drivers. In the DVE file, you can:

- Instantiate verification drivers (§2.3):
 - EVE drivers for co-simulation
 - EVE monitors for trace or triggers
 - Transactors from EVE Vertical Solutions' catalog
 - User-developed drivers in transaction-based verification (user-developed transactors)

The term *driver* in this manual refers indiscriminately to software and hardware drivers and monitors.

- Instantiate clock drivers to connect primary clocks of the design (§2.4).



In addition to the above, the DVE file may be used to:

- Use specific Verilog-like statements for:
 - Signal assignment (`assign`): to force DUT input signals, create a top-level port for internal signal of the design, or interconnect clocks (§2.7).
 - Register declaration (`reg`): to provide runtime force access to an internal undriven signal of the DUT or of a driver (for example, a reset input pin) (§2.8).
- Declare static and dynamic triggers: up to 16 triggers, regardless of the type, and without any specific limitation for either type (§2.9).
- Instantiate simple Verilog logic which may prove useful when you want to connect your DUT to a specific environment (§2.10).

Finally, the following advanced features can also be declared in the DVE file:

- Optional frequency constraints for the system clocks of ZeBu.
- Optional frequency constraints for the primary clocks of the design.
- Connection of the synthesized clocks, including their frequency constraints.

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

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

- The instantiation of the appropriate co-simulation driver. The DUT top ports are connected according to their direction at the DUT interface.
- When some controlled clock signals have been declared in the **Clock Declaration** panel, they are added in the DVE file for connection to the ZeBu clock generator. If no clock is declared in the **Clock Declaration** panel, a controlled clock is automatically added in the DVE file but it is not connected to the design.

The **zCui** settings for the generation of the DVE file template are available in the **Environment** panel and are described in the *ZeBu-Server Compilation Manual* (§3.4.1.3 in Rev. C).

2.2 Syntax for hierarchical paths in the DVE file

2.2.1 RTL hierarchical paths

By default, the paths in the DVE file are regarded as EDIF paths. When the design is synthesized with **zFAST**, it is possible to declare RTL hierarchical paths in the DVE file by adding one of the following lines:

```
defparam rtlname = yes;  
defparam rtlname = true;
```

Notes:

- RTL paths are not supported for elements such as `zceiClockPort`, `zClockPort` and `zIceClockPort`.
- RTL path expressions are Verilog compliant.
- When a Verilog path includes special characters, the DVE file must be written with escaped characters, as described in Section 2.2.3.

2.2.2 Using identifiers

2.2.2.1 Simple identifiers

A simple identifier starts with a letter or `'_'` and contains letters, `'_'`, and digits. It is not case sensitive. Special characters can also be used in identifiers by escaping them, as described in Section 2.2.3.

This simple identifier is named `T_IDENTIFIER` in the general syntax described in Section 2.12.

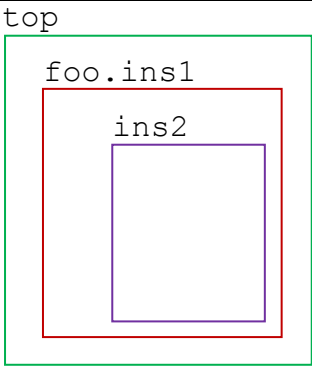
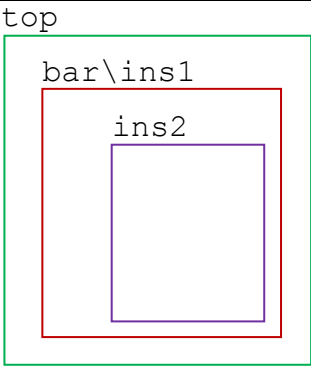
2.2.2.2 Hierarchical names

Just like in Verilog, hierarchical names are identifiers separated by the `'.'` separator. The first identifier is either of the following:

- The top signal of the DUT when a DUT signal needs to be accessed. For ex:
`top.ins1.ins2.signal`
- The name of a transactor's instance, for example when declaring internal signals of a transactor in the `SRAM_TRACE` instantiation for debugging purposes.

2.2.3 Using escaped characters in hierarchical names

When an instance name contains a dot or a backslash, these characters are interpreted as separators. To overcome this, any instance name containing such characters must be escaped as shown in the following examples:

	Example #1 (dot)	Example #2 (backslash)
		
Ordinary instance name	top.foo.ins1.ins2	top.bar\ins1.ins2
Escaped instance name	top.\foo.ins1<SPACE>.ins2	top.\bar\ins1<SPACE>.ins2

<SPACE>: Space, Tab, or Carriage Return

2.3 Declaring verification drivers

2.3.1 Instantiating the drivers

The verification drivers which are used with ZeBu are always instantiated in the DVE file and their interface is connected either to signals of the DUT or to fixed values:

```
<driver_name> <driver_instance> (
    .<driver_connector> ( <dut_signal_name> ),
    .<driver_connector> ( <value> ),
    [...]
);
```

Where:

- <driver_name>: name of the driver; the drivers provided in the ZeBu package are listed in Table 1.
- <driver_instance>: user-defined name for the driver instantiation in your environment.
- <driver_connector>: name of a port of the driver. Only part of the driver connector may be used.
- <dut_signal_name>: name of DUT signal (single-bit, vector or subpart of a vector).
- <value>: literal value either in binary (1'b0 for example) or hexadecimal (16'hF3E0 for example), on 1 or several bits.



If you instantiate the same driver several times in your environment, `driver_name` will be the same but `driver_instance` has to be different for each instantiation. You cannot have several instances with the same `driver_instance` in a DVE file.

For a vector connection, the signal name must contain the size of the vector to connect, either explicitly (e.g. `vect[31:0]` or `vect[15:8]` for partial connection) or implicitly (e.g. `vect[*]`). It is also possible to declare aggregate vectors using a list expression for the signal (e.g. `{vect[31:1], 1'b1}`), as shown in the general syntax described in Section 2.12.

The following drivers are provided in the ZeBu package:

Table 1: List of drivers provided in the ZeBu package

Driver Name	Usage
HDL_COSIM	HDL co-simulation or SystemC co-simulation. See ZeBu HDL Co-simulation Manual for details.
C_COSIM	Cycle-based, bit-accurate driver for C/C++ co-simulation. See ZeBu C++ Co-Simulation Manual for details.
MCKC_COSIM	Event-based, bit-accurate driver for C/C++ co-simulation. See ZeBu C++ Co-simulation Manual for details.
SRAM_TRACE	Hardware driver, enabling signals dumping at runtime. See ZeBu-Server Compilation Manual for details.
SMART_ZICE_ZSE	Specific driver for the Smart Z-ICE interface. See ZeBu-Server Smart Z-ICE Manual for details.

When integrating a transactor from EVE Vertical Solutions' catalog, all the information for a correct instantiation in the DVE file is available in the [Transactor User Manual](#) delivered with the transactor package.

2.3.2 Declaring specific parameters for a driver

Each driver can have several optional or mandatory parameters, according to the driver itself. The declaration of parameters MUST be after the instantiation of the driver in the DVE file (if not, the **Analyze DVE** step in **zCui** fails).

The declaration of driver-specific parameters has this syntax:

```
defparam <driver_instance>.<parameter_name> = <value>;
```

Where:

- `<driver_instance>`: Driver instantiation.
- `<parameter_name>`: Name of the parameter to be defined.
- `<value>`: String, name, decimal value, boolean value or anything acceptable for the corresponding parameter.
Note that boolean values can be any of the following: "yes", "true", "no" or "false".

For the co-simulation, trace and Smart Z-ICE drivers provided in the ZeBu package, the names of the parameters are predefined. Examples of DVE files with parameters for these drivers are available in Section 2.13.



In transaction-based verification, some predefined parameters exist to control the transaction-based environment, listed in Table 2 and some additional parameters can be specific for in the transactor itself.

Table 2: Predefined parameters for transaction-based verification

Parameter name	Value	Comments
use_edge_enables	<boolean_value>	If true, replaces registers of driver clocked by a zceiClockPort by registers clocked by a driverClk, with clock enable. Default: false
debug	<boolean_value>	If true, increases accessibility to internal signals of the driver at runtime; default: true
disable	<signal_identifier>	Specifies the name of a DVE signal which could be assigned at runtime to enable/disable a transactor. Typically this signal could have been declared as a register.
cclock	<clock_identifier>	Name of the clock signal that will be controlled by the zceiClockControl of the driver.

Example:

To improve accessibility to internal signals of the driver:

```
defparam my_xtor_inst.debug = true;
```

To control the activation of stream0 instantiation of the driver from a specific signal declared as a DVE register (user_xtor_disable):

```
reg user_xtor_disable;  
defparam stream0.disable = user_xtor_disable;
```



2.4 Declaring clock drivers

2.4.1 Declaring primary controlled clocks

Clock driver instantiation is used for the identification of primary clock signals during compilation (and optionally for the declaration of the DUT Reset signal). It connects the DUT with the outputs of a clock generation macro (`zceiClockPort`) which is used as follows:

```
zceiClockPort <my_clock_driver_name> (  
    .cclock(<my_clock_name>)  
    .creset(<my_reset_name>)  
);
```

Clock description (group allocation, waveform, virtual frequency) is done at runtime in the `designFeatures` file, as described in Section 3.4.

2.4.2 Declaring primary clocks coming from Direct ICE or Smart Z-ICE

Primary clock inputs coming from a Direct ICE or a Smart Z-ICE target are declared in the DVE file as instantiations of the `zIceClockPort` macro:

```
zIceClockPort <clock_ID> (.clock (<clock_name>))
```

Such clocks cannot be declared in the **zCui** graphical interface; therefore they are not added automatically when a DVE file template is generated. See more details in [ZeBu-Server Direct ICE Manual](#) and [ZeBu-Server Smart Z-ICE Manual](#).

2.4.3 Declaring frequency synthesizers for prototyping

For each FPGA module of the ZeBu-Server unit, two frequency synthesizers are available, programmable between 3.125 MHz and 450 MHz. These synthesizers can clock only instances in design FPGAs of the same module; in most cases, such clocks are used in one FPGA only because of the frequency constraints.

Such synthesizers can be used to generate asynchronous clocks to drive DCM in the design FPGAs or to clock the trace memory asynchronously.

Frequency synthesizers are declared in the DVE file as instantiations of `zClockPort` macros, with a maximum frequency constraint for compilation:

```
zClockPort <my_ClockPort_inst> (  
    .clock( <my_clock> )  
);  
defparam <my_ClockPort_inst>.frequency = <my_max_freq>;  
defparam <my_ClockPort_inst>.synthesis = true;
```

Where `<my_max_freq>` is the maximum frequency constraint, declared as a double-quoted string with value (integer) and unit (kHz or MHz, case insensitive). For ex: `<my_max_freq>="54000Khz"`.

Notes:

- Their actual frequency can be modified at runtime in the `designFeatures` file, as long as it is lower than `<my_max_freq>`.
- The skew for such clocks can be controlled from **zCui** by selecting **Asynchronous** in the **Skew Offset** frame of **Clock Handling** panel. The value of the inserted delay (indicated in ns) should be set between 100 and 640.

2.5 Declaring clocks parameters for ZeBu system clocks

The appropriate syntax for the declaration of clock parameters is the following:

```
defparam <clock_name> = <frequency>
```

Where:

- <clock_name>: any clock parameter listed in.
- <frequency>: string with an integer value and a mandatory unit (MHz or kHz, case sensitive), as described in the BNF available in Section 2.12.

**Table 3: ZeBu System Clocks
with frequency constraints for compilation**

<clock_name>	Range of values	Default value
masterfrequency	50-160 MHz	100 MHz
driverfrequency	1-100 MHz	30 MHz
xclkfrequency	200-450 MHz	400 MHz

Example:

```
defparam driverfrequency = "60 MHz";
```

The frequency values declared in the DVE file for ZeBu system clocks constrain the FPGA compilation. Declaring a frequency smaller than the default value may help solve ISE compilation problems for a given FPGA by reducing the actual timing constraints for this FPGA.

Note that it is not possible to declare for runtime (in the designFeatures file) a frequency higher than the one declared for compilation in the DVE file.

Example:

To run the emulation with xclk frequency set between 200 and 425 MHz, one of the following lines should be added in the DVE file for compilation:

```
defparam xclkfrequency = "425 MHz";
```

which is equivalent to:

```
defparam xclkfrequency = "425000 kHz";
```

2.6 Declaring parameters for advanced clock modeling

2.6.1 Modifying the number of available controlled clocks

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

Number of controlled primary clocks declared in the DVE file	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Maximum number of controlled clocks supported by the ZeBu clock generator	2	2	4	4	8	8	8	8	16	16	16	16	16	16	16	16
Possible number of additional clocks	1	0	1	0	3	2	1	0	7	6	5	4	3	2	1	0

The following line in the DVE files modifies the default number of controlled clocks that can be connected as primary design clocks:

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




2.6.2 Clock generator accuracy

To support a wider range of frequencies for primary clocks from the ZeBu clock generator, it is possible to declare a 32-bit accuracy in the DVE file instead of the default 24-bit accuracy:

```
defparam fk_clockgen_accuracy = 32 ;
```

Notes:

- This declaration is automatically propagated for runtime.
- The 32-bit accuracy may impact the duration of FPGA Place and Route for IF FPGA, or even the capability to compile it.
- If the 32-bit accuracy is not declared in the DVE file, the default 24-bit accuracy prevails.

2.7 Using assign statements

assign statements in the DVE file enable you to optimize the use of limited resources such as clocks, or to optimize a HW/SW interface, or to assign a fixed value to a DUT signal. The usual syntax for such statements is:

```
assign <signal_i> = <signal_j>
```

Where signals can be scalar or vectors, and <signal_j> can be a list expression.

Only scalars/vectors are supported on left-hand side, not list expressions.

It is possible to use a simple form of Verilog conditional assignment in the DVE file where only signal names would be used (no complex expressions).

Examples:

If clock1 is declared as the output of a clock port and you need an identical clock, clock2, you can physically connect clock2 to clock1 to save clock resources:

```
assign clock2 = clock1;
```

If you need to transmit two reset signals across the HW/SW interface, you can connect them together and the ZeBu compilation tools will keep only one physical reset signal at the interface:

```
assign reset2 = reset1;
```

To force y[31:0] vector to the first 31 bits of counter and then 1 as the LSB, you can use a list expression on the right side of assignation:

```
assign y[31:0] = { counter[31:1] , 1'b1 };
```

Example of a conditional assignment in the DVE file, in which chk_in[8] is either cnt_out[15] or cnt_out[16] according to the value of select:

```
assign chk_in[8] = select ? cnt_out[15] : cnt_out[16];
```



2.8 Declaring registers

Declaring a register in the DVE file creates a programmable input which can be connected to an input port of the DUT or of a transactor, or it can be used as an additional control signal, with an `assign` statement or with `defparam` declarations.

This signal can be written at runtime from the **Monitor** function of **zRun** (see [zRun Emulation Interface Manual](#), § 3.3.5 in Rev. D) or from a C/C++ testbench.

The syntax for register declaration is:

```
reg <signal>;
```

Where `<signal>` can be a single-bit signal or a vector.

In case of a vector, the syntax is:

```
reg <range> <vector>;
```

2.9 Declaring triggers

In the DVE file, you can declare up to 16 triggers whatever their types, without any specific limitation for each type.

These triggers can be used either in a C/C++ testbench or in **zRun**. Detailed information about the use of triggers with **zRun** is available in the [ZeBu zRun Emulation Interface Manual](#).

There are two types of triggers:

- Static triggers to compare two signals or to compare a signal with a constant.
- Dynamic triggers to test the value of one or more signals against non-predefined constants. Only the list of signals is declared in the DVE file; the actual trigger expression is programmed during emulation.

2.9.1 Static triggers

The syntax to declare a static trigger is:

```
trigger <trigger_name> = <trigger expression>
```

See BNF of DVE file in Section 2.12 for detailed syntax of `trigger_expression`. A static trigger defines one interface signal in the DVE file.

Example:

The following static trigger will be activated when `counter[0]` is 1.

```
trigger bit_L_counter = (counter[0] == 1'b1);
```

2.9.2 Dynamic triggers

Dynamic triggers are declared in the DVE file as an instantiation of the `zceiTrigger` driver, with the following syntax:

```
zceiTrigger <name_trigger>(.output_bin(<signal_1>, <signal2>);
```

The `zceiTrigger` driver has an `output_bin` port and the signals connected to this port are available at runtime to program the trigger expression in **zRun** or in the C/C++ testbench.



Example:

With the following declaration, a dynamic trigger can be programmed at runtime using the 16 bits of `adder[15:0]`:

```
zceiTrigger dyn_adder (.output_bin({adder[15:0]}));
```

2.10 Instantiating simple Verilog logic gates

The syntax to be used to instantiate a Verilog gate primitive depends on the type of gate primitive which needs to be instantiated:

```
<gate_specifier> <instance_name> ( <gate_cnx_lst> );
```

Where:

- `<gate_specifier>` is one of the following gate types:
and, nand, or, nor, xnor, buf, not, bufif0, bufif1, notif0, notif1
- (`<gate_cnx_lst>`) depends on the gate type:
 - o and, nand, or, nor, xnor: (out, a, b)
 - o buf, not: (out, in)
 - o bufif0, bufif1, notif0, notif1: (out, in, en)

Example:

```
notif1 ins_foo3 ( out, in, en );
```

Restriction: Instantiation is supported only on one-bit wide signals. For example, to perform a bit-by-bit and operation between `a[3:0]` and `b[3:0]` in `out[3:0]`, you should use the following statements:

```
and ins_and0 ( out[0], a[0], b[0] );  
and ins_and1 ( out[1], a[1], b[1] );  
and ins_and2 ( out[2], a[2], b[2] );  
and ins_and3 ( out[3], a[3], b[3] );
```

Limitation: Arithmetic expressions such as "A & B" are not supported.



2.11 Sampling on an uncontrolled clock

By default, the sampling clock for SRAM trace is not declared for compilation and can be selected at runtime: it can be either driverClock or any primary controlled clock.

If another clock is expected for sampling, it has to be declared with `.sample_clk` connection in the instantiation of the SRAM Trace driver in the DVE file. This is actually useful to sample on an uncontrolled clock either a frequency synthesizer or a clock from the Smart Z-ICE of Direct ICE interfaces. An additional parameter must be set for the edge active for sampling.

```
SRAM_TRACE <sram_trace_inst> (  
    .sample_clk (<sampling_clock>);  
    .output_bin ( [...]);  
);  
defparam <sram_trace_inst>.edge=<clock_edge>;
```

Where:

- `<sampling_clock>` can be any signal declared in the DVE file, in particular a primary clock of the design (`zceiClockPort`) an output of frequency synthesizer (`zClockPort`) or an input clock from the Smart Z-ICE or Direct ICE interfaces (`zIceClockPort`).
- `<clock_edge>` is the active edge for sampling. It can be one of the following values: `pos`, `neg` or `both`.

If this sampling clock is declared in the DVE file, it cannot be changed at runtime through the `designFeatures` file.

A typical use is to sample on the Smart Z-ICE input clock when a JTAG debugger is connected to the Smart Z-ICE interface.

Example:

```
SRAM_TRACE sram_trace_0 (  
    .sample_clk( clk ),  
    .output_bin ({rstn, ena, cnt[31:0], counter_A, counter_BB,  
                 counter_1000, counter_10000, cnt_2[31:2], cnt_3[24:6],  
                 counter_10000000, bit_4, bit_10, bit_16  
});  
defparam sram_trace_0.edge = pos ;
```



2.12 General syntax

The DVE file syntax is very similar to Verilog and its BNF description is as follows:

```
dve_list :  
    /*empty */  
    | dve_list  dve_item  
  
dve_item :  
    driver_instantiation  
    | driver_defparam  
    | global_defparam  
    | trigger_definition  
    | assignment_declaration  
    | register_declaration  
    | gate_instantiation  
  
driver_instantiation :  
    driver_name instance_name '(' list_of_driver_connections  
    ')' ';'   
  
list_of_driver_connections :  
    | port_connection  
    | list_of_driver_connections ',' port_connection  
  
port_connection :  
    '.' port_name '(' signal_expression ')'  
  
driver_defparam :  
    defparam signal_path parameter_name '=' alphanum ';'   
signal_path :  
    signal_name '.'  
    | signal_path signal_name '.'  
  
global_defparam :  
    defparam parameter_name '=' alphanum ';'   
  
signal_expression :  
    signal_name  
    | signal_name '[' index ']'   
    | signal_name '[' index ':' index ']'   
    | constant_expression  
    | '{' list_of_signal_expressions '}'   
  
list_of_signal_expressions :  
    /* empty */  
    | signal_expression  
    | list_of_signal_expressions ',' signal_expression  
  
constant_expression :  
    T_BINARY_CONSTANT  
    | T_HEXA_CONSTANT
```



```
trigger_definition :
    trigger_declaration '=' trigger_expression ';'

trigger_declaration :
    trigger trigger_name

trigger_expression :
    unary_expression
    | trigger_expression '&' unary_expression
    | trigger_expression '|' unary_expression
    | trigger_expression '^' unary_expression
    | '~' '(' trigger_expression ')'

unary_expression :
    '(' trigger_expression ')'
    | signal_expression op_compar signal_expression
    | signal_attribute_expression
    | trigger_name

op_compar :
    '=='
    | '!='

signal_attribute_expression :
    posedge signal_expression
    | negedge signal_expression

alphanum :
    T_IDENTIFIER
    | T_NUMBER
    | T_STRING

assignment_declaration :
    assign signal_expression '=' signal_expression ';'
    | assign signal_expression '=' signal_expression '?'
      signal_expression ':' signal_expression ';'

gate_instantiation :
    gate_specifier gate_ins_lst ';'

gate_ins_lst :
    gate_instance
    | gate_ins_lst ',' gate_instance

gate_instance :
    /* empty */ '(' gate_cnx_lst ')'
    | instance_name '(' gate_cnx_lst ')'

gate_cnx_lst :
    signal_expression
    | gate_cnx_lst ',' signal_expression

gate_specifier :
```



```
and
| nand
| or
| nor
| xor
| xnor
| buf
| not
| bufif0
| bufif1
| notif0
| notif1

register_declaration :
    reg '[' index ':' index ']' list_of_reg_names ';'
    | reg list_of_reg_names ';'

list_of_reg_names :
    /* empty */
    | reg_name
    | list_of_reg_names ',' reg_name

index :
    T_NUMBER
```

Where:

- trigger_name, reg_name, driver_name, instance_name, port_name, signal_name, parameter_name, FPGA_name: **identifiers** (T_IDENTIFIER) starting with a letter or '_' and containing letters, '_' and digits. Identifiers are not case sensitive. Special characters can also be used in identifiers by escaping them, as described in Section 2.2.3.
- T_NUMBER: decimal integer containing only digits.
- T_STRING: alphanumerical string starting/ending with '"' (e.g. "ab123").
- T_BINARY_CONSTANT: sized binary constant similar to Verilog binary constants. The corresponding regular expression is:
`{number}\{'{b_base}({binary_digit})+`

Where: b_base: b or B

binary_digit: 0 or 1

Note that blank characters are not allowed.

- T_HEX_CONSTANT: sized hexadecimal constant similar to Verilog hexadecimal constants. The corresponding regular expression is:

`{number}\{'{h_base}({hexa_digit})+`

Where: hexa_digit: [0..9], [A..F], [a..f]

h_base: h or H

Note that blank characters are not allowed.

- Comments can be inserted in the DVE file using C/C++ style:
 - /* comment */
 - // comment

2.13 Examples

2.13.1 DVE file instantiating an HDL_COSIM driver

An HDL_COSIM driver instantiation must include some of these blocks:

- .input_bin ({...}),
- .input_tri ({...}),
- .output_bin ({...}),
- .output_tri ({...}),
- .inout_tri ({...})

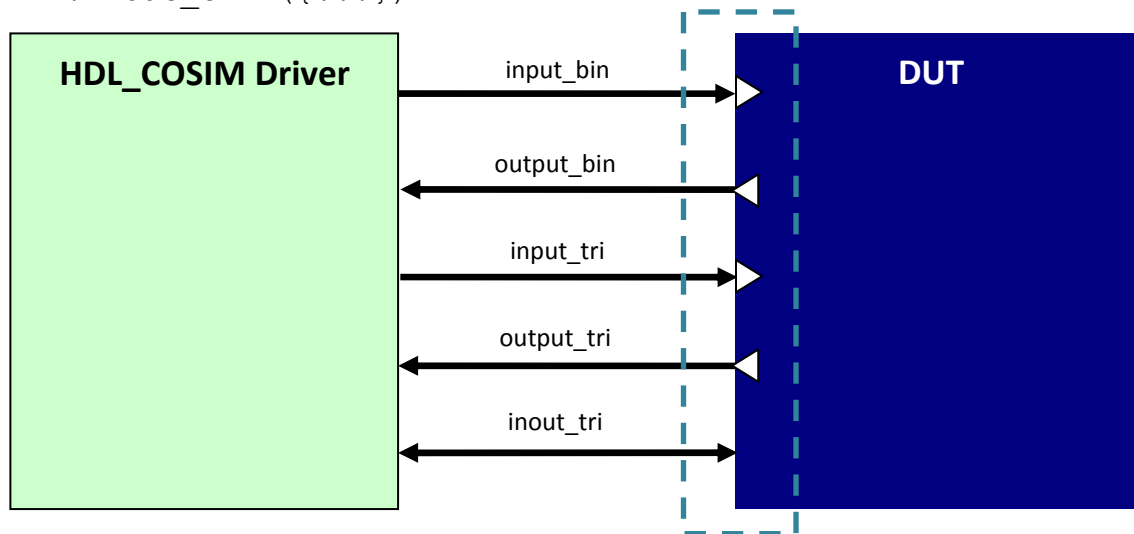


Figure 2: Instantiation of an HDL_COSIM driver in the DVE file

Example:

```
HDL_COSIM my_hdlcosim (
  .input_bin ({ d, d2
  }),
  .input_tri ({ bus_in, bus_in2
  }),
  .output_bin ({ q, q2
  }),
  .output_tri ({ bus_out, bus_out2
  }),
  .inout_tri ({ io, io2
  }));

defparam my_hdlcosim.clock_0 = clk;
zceiClockPort clkport (
  .cclock( clk )
);
```

See dedicated chapter of the *[ZeBu HDL Co-simulation Manual](#)* for details on the instantiation of HDL_COSIM driver in the DVE file.

2.13.2 DVE file instantiating a C_COSIM driver

A C_COSIM driver instantiation must include some of the following blocks:

- .input_bin ({...})
- .input_tri ({...})
- .output_bin ({...})
- .output_tri ({...})
- .inout_tri ({...})

The C_COSIM driver clock must be specified with the cclock parameter:

- clock name: name of the signal connected to cclock

Optional declarations: Static triggers and dynamic triggers

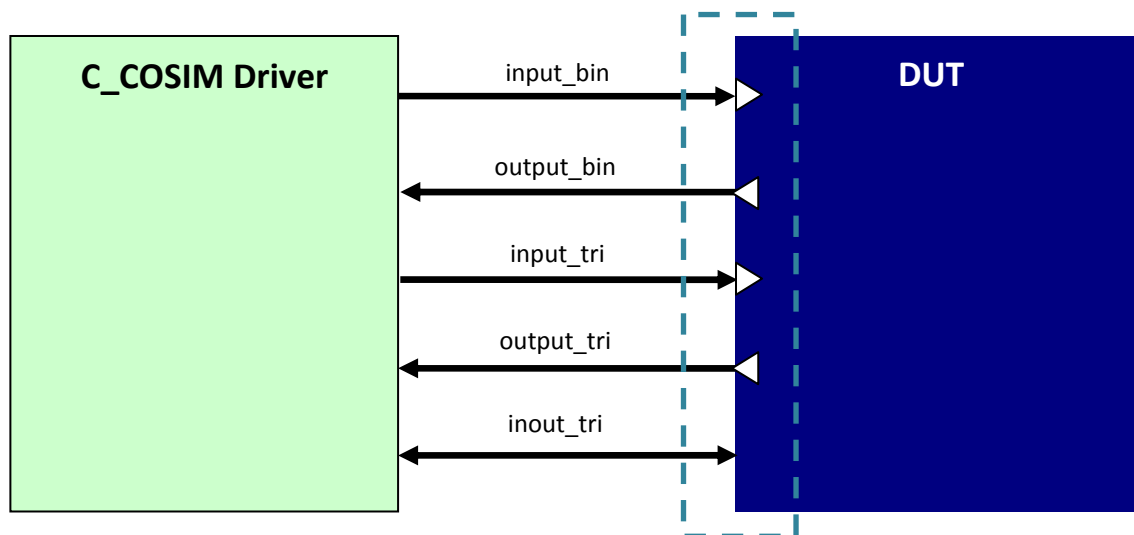


Figure 3: Instantiation of a C_COSIM driver in the DVE file

Example:

```
C_COSIM dut_ccosim (
  .input_bin({
    resetn,
    cst[15:0]
  }),
  .output_bin({
    counter[15:0],
    adder[15:0],
    dout[15:0],
    bit_L_adder,
    bit_M_adder,
    bit_H_adder,
    adder_mod15,
    dyn_adder,
    dyn_L_adder,
    dyn_M_adder,
    dyn_H_adder,
    bit_L_counter,
```




```
    bit_M_counter,  
    bit_H_counter,  
    counter_mod15,  
    dyn_counter,  
    dyn_H_counter,  
    dyn_L_counter,  
    dyn_M_counter  
));  
  
defparam dut_ccosim.cclock = "clock";  
  
zceiClockPort clock_ClockPort0 (  
    .cclock( clock )  
);  
  
trigger bit_L_adder = (adder[0]    == 1'b1);  
trigger bit_M_adder = (adder[8]    == 1'b1);  
trigger bit_H_adder = (adder[15]   == 1'b1);  
trigger adder_mod15 = (adder[3:0]  == 4'b1111);  
zceiTrigger dyn_adder    (.output_bin({adder[15:0]}));  
zceiTrigger dyn_L_adder  (.output_bin({adder[7:0]}));  
zceiTrigger dyn_M_adder  (.output_bin({adder[11:4]}));  
zceiTrigger dyn_H_adder  (.output_bin({adder[15:8]}));  
  
trigger bit_L_counter = (counter[0]    == 1'b1);  
trigger bit_M_counter = (counter[8]    == 1'b1);  
trigger bit_H_counter = (counter[15]   == 1'b1);  
trigger counter_mod15 = (counter[3:0]  == 4'b1111);  
zceiTrigger dyn_counter  (.output_bin({counter[15:0]}));  
zceiTrigger dyn_L_counter (.output_bin({counter[7:0]}));  
zceiTrigger dyn_M_counter (.output_bin({counter[11:4]}));  
zceiTrigger dyn_H_counter (.output_bin({counter[15:8]}));
```

See dedicated chapter of the [ZeBu C++ Co-simulation Manual](#) for details on the instantiation of C_COSIM driver in the DVE file.

2.13.3 DVE file instantiating an SRAM_TRACE driver

An SRAM_TRACE driver instantiation must include this single block:

- `.output_bin({...}),`

Optional declarations:

- New clock name (default name: name of the signal connected to `cclock`).
- Assign statements

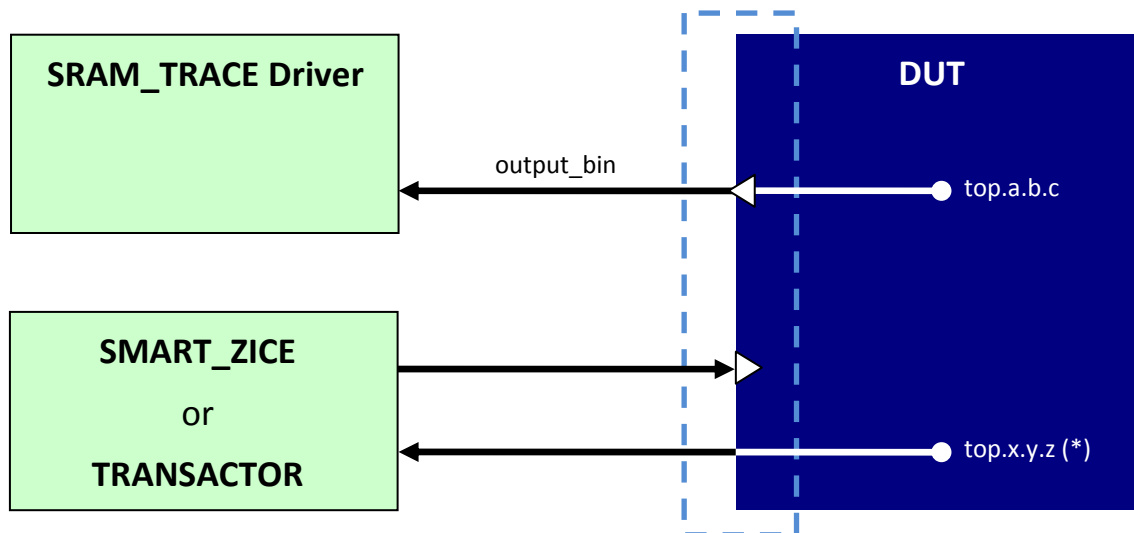


Figure 4: Instantiation of an SRAM_TRACE driver in the DVE file

(*) Hierachical references: To proceed with transactor debugging, you can declare some internal signals of a transactor with the instantiation of the SRAM_TRACE driver in the DVE file. For example: `top.ins1.ins2.signal`

Example:

```
SRAM_TRACE sram_trace_0 (
    .output_bin ({
        resetn,
        cst[15:0],
        counter[15:0],
        adder[15:0],
        dout[15:0],
        bit_L_adder,
        bit_M_adder,
        bit_H_adder,
        adder_mod15,
        dyn_adder,
        dyn_L_adder,
        dyn_M_adder,
        dyn_H_adder,
        bit_L_counter,
        bit_M_counter,
        bit_H_counter,
        counter_mod15,
        dyn_counter,
```



```
    dyn_H_counter,  
    dyn_L_counter,  
    dyn_M_counter  
  })  
);  
  
defparam sram_trace_0.cclock = clock;  
  
assign adder[0]    = dut.adder_0.dout[0]    ;  
assign adder[1]    = dut.adder_0.dout[1]    ;  
assign adder[2]    = dut.adder_0.dout[2]    ;  
assign adder[3]    = dut.adder_0.dout[3]    ;  
assign adder[4]    = dut.adder_0.dout[4]    ;  
assign adder[5]    = dut.adder_0.dout[5]    ;  
assign adder[6]    = dut.adder_0.dout[6]    ;  
assign adder[7]    = dut.adder_0.dout[7]    ;  
assign adder[8]    = dut.adder_0.dout[8]    ;  
assign adder[9]    = dut.adder_0.dout[9]    ;  
assign adder[10]   = dut.adder_0.dout[10]   ;  
assign adder[11]   = dut.adder_0.dout[11]   ;  
assign adder[12]   = dut.adder_0.dout[12]   ;  
assign adder[13]   = dut.adder_0.dout[13]   ;  
assign adder[14]   = dut.adder_0.dout[14]   ;  
assign adder[15]   = dut.adder_0.dout[15]   ;  
assign counter[0]  = dut.counter_0.dout[0]  ;  
assign counter[1]  = dut.counter_0.dout[1]  ;  
assign counter[2]  = dut.counter_0.dout[2]  ;  
assign counter[3]  = dut.counter_0.dout[3]  ;  
assign counter[4]  = dut.counter_0.dout[4]  ;  
assign counter[5]  = dut.counter_0.dout[5]  ;  
assign counter[6]  = dut.counter_0.dout[6]  ;  
assign counter[7]  = dut.counter_0.dout[7]  ;  
assign counter[8]  = dut.counter_0.dout[8]  ;  
assign counter[9]  = dut.counter_0.dout[9]  ;  
assign counter[10] = dut.counter_0.dout[10];  
assign counter[11] = dut.counter_0.dout[11];  
assign counter[12] = dut.counter_0.dout[12];  
assign counter[13] = dut.counter_0.dout[13];  
assign counter[14] = dut.counter_0.dout[14];  
assign counter[15] = dut.counter_0.dout[15];
```

See *[ZeBu Compilation Manual](#)* for details on the instantiation of SRAM_TRACE driver in the DVE file.



3 designFeatures file

3.1 Introduction

The `designFeatures` file is an optional file for the ZeBu runtime software. This file contains user settings which override the default values for emulation runtime.

The `designFeatures` file contains in particular the following elements:

- Declaration of processes (§3.3)
- Parameters for design clocks (§3.4)
- Parameters for transactors (§3.5)
- Initialization information for memories (§3.6)
- Declaration of System clock frequencies (§3.7)
- Relocation declaration for Smart Z-ICE (§3.11)

The path to the `designFeatures` file can be provided as an argument when the `open` method is called in the testbench. However, if a file called `designFeatures` exists in the directory where the runtime software is launched or in compilation output directory (usually `zebu.work`), this file is automatically used by the ZeBu runtime software.

If not found in any of the above, no `designFeatures` file is loaded and all default values are kept for emulation runtime.

3.1.1 Syntax rules

The parameter names in the `designFeatures` file are case insensitive, but all file and signal names are case sensitive.

Commented lines start with a `#` character.

3.1.2 `designFeatures.<PCname>.help`

This file is an output template file generated in the current directory and containing:

- user-defined settings declared in the `designFeatures` file (if present) and default settings for the other parameters.
- only default settings when the `designFeatures` file is not present.

The `designFeatures.<PCname>.help` file can be modified and saved as `designFeatures` which can later be used in another emulation runtime session.

3.2 Location of the calibration files

When the calibration files are not located in the default directory (`$ZEBU_SYSTEM_DIR/calibration`), it is possible to declare the path to the calibration files in the `designFeatures` file:

```
$calibrationPath = "<path>";
```



3.3 Declarations of processes

3.3.1 Single-process environment

When the runtime environment is based on a single process, there is no need to declare the process name in the designFeatures file: default_process will be used automatically. This default configuration can be seen in the resulting designFeatures.help file:

```
$nbProcess = 1;  
$process_0 = "default_process";
```

3.3.2 Multi-process environment

In a multi-process environment, the declaration of the processes is mandatory:

```
$nbProcess = <nb_process>;  
$process_<ID> = "<process_name>";
```

Where the maximum number of processes is 16:

- <nb_process> is an integer between 1 and 16
- <ID> is an integer index between 0 and 15 (processes must be declared with contiguous indexes starting from 0).
- <process_name> is a user-defined string which is the identifier of the process in the open method and is displayed in the log files.

Example:

```
$nbProcess = 1;  
$process_0 = "myProcess";
```

In the testbench

```
Board *board = NULL;  
// open session  
if ((board = Board::open ("myZebuWork", "designFeatures",  
"myProcess")) == NULL)  
    exit (1);
```

```
-- ZeBu : cpp_test_bench : Looking for a connection (pid 28200 at Tue 3 8 2010 - 17:46:11)  
-- ZeBu : cpp_test_bench : "my_process" is a full-capability process working on  
    "../zebu.work".
```

3.3.3 Unidentified processes

A C/C++ testbench can be executed without being explicitly declared as a process if it does not control any co-simulation driver or transactor. Such a process is known as a control-process and can be declared in the open method without being declared in the designFeatures file.

A control process is mostly intended to investigate or control clocks, memories, logic analyzers, triggers, signals, etc. They cannot be used to control top-level IOs.



However clocks declared in the DVE file with a `zceiClockPort` can only be driven by identified processes.

Example:

```
$nbProcess = 2;  
$process_0 = "bench_0";  
$process_1 = "bench_1";
```

In the testbench:

```
Board *board = NULL;  
// open session  
if ((board = Board::open ("myZebuWork", "designFeatures",  
"bench_2")) == NULL)  
    exit (1);
```

```
-- ZeBu : tb_dut : WARNING : A process name has been specified "bench_2" at open time, but it  
                        is not specified in the "./designFeatures" file.  
-- ZeBu : tb_dut : WARNING :      The list of specified process names is :  
-- ZeBu : tb_dut : WARNING :      "bench_0"  
-- ZeBu : tb_dut : WARNING :      "bench_1"  
-- ZeBu : tb_dut : Looking for a connection (pid 11307 at Wed 9 3 2011 - 09:20:51) ...  
-- ZeBu : tb_dut : "bench_2" is a control-only process working on "../zcui.work/zebu.work".
```

3.3.4 When initialization phases not carried out on memory

When a control-only C/C++ testbench, some initialization phases will not be carried out, namely on memory. In these circumstances, the design will not work even if no error is generated. To avoid this, user must identify the testbench in the `designFeatures` file in the same way as it is done for multi-process.

```
$nbProcess = 1;  
$process_0 = "<process_name>";
```

3.3.5 zRun unidentified process

zRun is a particular case of unidentified processes. You cannot connect with **zRun** in stand-alone mode using a design with transactor declarations. Connection will fail with the following error message:

```
-- ZeBu : zRun : ERROR : LUI2007E : In this configuration, zRun should be used with an  
                        embedded test bench.  
-- ZeBu : zRun : ERROR : LUI2008E : Ports has been detected, they were declared at  
                        compilation time.  
-- ZeBu : zRun : ERROR : You can try to force "$nbProcess = 0;" in the designFeatures file.  
-- ZeBu : zRun : ERROR : ZHW0909E : Cannot open Zebu: an error occurred during connection.
```

Connection will succeed if you set `nbProcess` to 0 in the `designFeatures` file:

```
"$nbProcess = 0;"
```



3.4 Declarations for Design Clocks

3.4.1 Declaring Additional Clocks

You can create a dummy clock, which may be useful for Trace or Monitoring oversampling. The appropriate syntax is the following:

```
$newClock = "U0.M0.<my_dummy_clock>";
```

<my_dummy_clock> in the same way as usual DUT clocks as described in Section 3.4.2.

When a new dummy clock is successfully added, this message is displayed:

```
-- ZeBu : zServer : The new clock "U0.M0.d_clk1" will be mapped on clock 1.
```

Note that this clock is included in the maximum number of clocks provided by the clock generator.

If the maximum number of primary clocks has been exceeded and an additional clock is declared, the following error message is displayed:

```
-- ZeBu : zServer : ERROR : LUI1433E : This is a 2 clocks generator.  
-- ZeBu : zServer : ERROR :           : You cannot register more than 1 new user clock  
   on this clock generator.  
-- ZeBu : zServer : ERROR : There is already 1 user clock.
```

The maximum number of clocks supported by the clock generator can be modified in the DVE file, as described in Section 2.6.1. After this modification in the DVE file, it is mandatory to recompile the design.

3.4.2 Parameters for primary clocks

3.4.2.1 Syntax

The following settings allow you to define clock parameters. Each clock is described in the designFeatures file with its own parameters, and it can be linked to other clocks in a group. The syntax for the declaration of clock parameters is the following (explicit description):

```
$U0.M0.<my_clock>.Mode = "controlled | free-running | in-situ";  
$U0.M0.<my_clock>.Waveform = "_-";  
$U0.M0.<my_clock>.Frequency = <my_realFreq>;  
$U0.M0.<my_clock>.VirtualFrequency = <my_virtFreq>;  
$U0.M0.<my_clock>.GroupName = "<my_group>";  
$U0.M0.<my_clock>.Tolerance = "no | yes";
```

Where:

- Mode is "controlled" (clock defined as zceiClockPort) when no Mode is defined. Declaring a clock as "in-situ" is optional if it was specified in the DVE file as an *in-situ* clock (i.e. an input clock from the Smart Z-ICE clock connectors). For details on appropriate clock declarations for the Smart Z-ICE interface, refer to the [ZeBu Smart Z-ICE Manual](#).
- Waveform is a sequence of characters ("[_-]+"). Character sequences should always start with "_". Default is "_-".
- Frequency is mandatory for free-running clocks and is not applicable for other types of clocks. <my_real_freq> is the frequency in kHz (float value).



- VirtualFrequency is a ratio between frequencies of clocks belonging to the same clock group. It is only applicable for controlled clocks. Default is 1
- GroupName identifies a clock group. By default, 2 given clocks will always belong to the same group. Use this parameter to split them in separate groups. Default is "MyGroup".
- Tolerance: when several primary clocks provided by the ZeBu clock generator are declared for runtime in a single clock group, the Tolerance parameter optimizes the runtime performance when the order of clock edges is not relevant and only the median frequency and the frequency ratios matter. See details in Section 3.4.2.5.

3.4.2.2 Example of DUT clock description in controlled mode

When a clock group contains a single clock, there is no need to declare anything to describe a clock in controlled mode (default mode).

When a clock group contains more than one clock, you must declare a frequency ratio through the VirtualFrequency parameter or the Waveform parameter. Since the clocks are provided by the ZeBu clock generator, ZeBu guarantees that a frequency is set as a relative value (VirtualFrequency) or as a Waveform.

To declare a frequency ratio of 2 through the VirtualFrequency parameter:

```
$U0.M0.clock1.Mode = "controlled"
$U0.M0.clock1.Waveform = "_-"
$U0.M0.clock1.VirtualFrequency = 1;
$U0.M0.clock1.GroupName = "clock_group"

$U0.M0.clock2.Mode = "controlled"
$U0.M0.clock2.Waveform = "_-"
$U0.M0.clock2.VirtualFrequency = 2;
$U0.M0.clock2.GroupName = "clock_group"
```

To declare a frequency ratio of 2 through the Waveform parameter:

```
$U0.M0.clock1.Mode = "controlled"
$U0.M0.clock1.Waveform = "_-"
$U0.M0.clock1.VirtualFrequency = 1;
$U0.M0.clock1.GroupName = "clock_group"

$U0.M0.clock2.Mode = "controlled"
$U0.M0.clock2.Waveform = "_-_"
$U0.M0.clock2.VirtualFrequency = 1;
$U0.M0.clock2.GroupName = "clock_group"
```

3.4.2.3 Example of DUT clock description in free-running mode

The clocks are provided by the ZeBu clock generator, and the frequency is fixed.

```
$U0.M0.my_clock.Mode = "free-running";
$U0.M0.my_clock.Frequency = 5000;
```

3.4.2.4 Example of DUT clock description in in-situ mode

The clocks are provided externally, and the frequency cannot be set.

```
$U0.M0.my_clock.Mode = "in-situ";
```




3.4.2.5 Optimizing the runtime performance with Tolerance

This parameter can be declared to improve the achievable runtime performance when several primary clocks belonging to the same clock group are declared for runtime. The performance improvement is more visible when:

- the number of primary clocks is larger.
- the declared waveforms and ratios are heterogeneous.
- the clock edges which are very close to one another are processed simultaneously, on a group-by-group basis or on a clock-by-clock basis.

To activate the Tolerance parameter, the following line(s) must be added for each group/clock which needs to be modified:

```
$U0.M0.<group>.Tolerance=<value>  
$U0.M0.<clock>.Tolerance=<value>
```

Where <value> is a string with double quotes and with the following values:

- <value>="no": Default
- <value>="yes": clock edges which are very close to one another are merged.

Notes:

- Tolerance can only be used with controlled clocks.
- At least 2 clocks are needed in the same group for an effective calculation.
- Only yes/no attributes are accepted and they are case insensitive.

When the tolerance set for a given clock group is different from the tolerance set for a single clock in this group, all the clocks belonging to the group will have the group tolerance except that clock, as shown in the example below:

```
#Group setting for group1  
$U0.M0.group1.Tolerance = "yes";  
  
#CLK0 uses the group setting for tolerance  
$U0.M0.CLK0.VirtualFrequency = 8;  
$U0.M0.CLK0.GroupName = "group1";  
  
# CLK1 overwrites the group setting to disable tolerance  
$U0.M0.CLK1.VirtualFrequency = 5;  
$U0.M0.CLK1.GroupName = "group1";  
$U0.M0.CLK1.Tolerance = "no";  
  
# CLK2 uses the group setting for tolerance.  
$U0.M0.CLK2.VirtualFrequency = 2;  
$U0.M0.CLK2.GroupName = "group1";
```

If a clock and a clock group have similar names in the designfeatures file, that causes ambiguity in the declaration of tolerance and runtime exits in error as follows:

```
"LUI1990E" : "Ambiguous designfeatures declaration : $U0.M0.clk.Tolerance = "yes"  
"A Clock Name and a Group Name are the same (clk) : "Tolerance" can not be attributed"
```



3.4.2.7 Clock declaration for HDL co-simulation

In HDL co-simulation, you can define the `designFeatures` file explicitly by using the `-designFeatures` option when launching your HDL simulator. In HDL co-simulation, the clocks controlled by the HDL testbench must be declared without being attached to a group in the `designfeatures` file. Additional design clocks which are not controlled from the HDL testbench can be grouped.

3.4.2.8 Clock declaration for C/C++ co-simulation

For C/C++ co-simulation, you can describe a DUT clock in the C/C++ testbench instead of the `designFeatures` file. See the [ZeBu C++ Co-Simulation Manual](#) for details. However, for restore operations EVE recommends that the user declares clock parameters from the `designfeatures` file only.

3.4.3 Declaration with a separate Clock File

For an easier reading of the `designFeatures` file, the same syntax can be declared in a separate clock file, as described in Chapter 4. This clock file is declared in the `designFeatures` file, as shown below:

```
$U0.M0.<clock_name>.File = "my_clock_file";
```

3.4.4 Propagation delay

The synchronizers inserted on clock paths mapped into FK are programmable:

```
$FKpropagationDelay = <my_FKpropagationDelay>;
```

Where `my_FKpropagationDelay` is an integer between 0 and 15 (default is 2), corresponding to a number of system clock periods.

When the `$FKpropagationDelay` is present but without synchronizers inserted by compilation (no clock mapping in FK), the runtime setting is ignored and the following message is displayed:

```
$FKpropagationDelay specification is ignored (synchronizers are not used)
```

3.4.5 Offset delay for clock skews

A delay can be inserted at runtime to offset the clock skews. It is programmed with 2 parameters, `zClockFilterTime` and `zClockSkewTime`, which must be set to control a glitch filtering algorithm.

- `zClockFilterTime` (ns) is the delay of the longest path from a clock cone entry to a filter. The filters are locked within this delay and all filters are opened at the same time after the specified delay.
- `zClockSkewTime` (ns) = `zClockFilterTime` + Δ
 Δ = delay of the longest path from a filter to a sequential element

Note: `zClockFilterTime` is always smaller than `zClockSkewTime`. When static timing analysis is used, **zTime** estimates both parameters and optionally transmits them to runtime with the `driverClock` frequency.



```
$zClockSkewTime = <my_skew>;  
$zClockSkewSel = "driverClk | allUserClk | userClkMask=<mask>";  
$zClockSkewMode = "pulse | level";
```

```
$zClockFilterTime = <my_filter>;  
$zClockFilterSel = "driverClk | allUserClk | userClkMask=<mask>";  
$zClockFilterMode = "pulse | level";
```

Where:

- <my_skew> and <my_filter> are integers between 0 and 255 corresponding to a number of system clock periods (default is compilation value: 40).
- <mask> is a 16-bit integer pattern that will enable the 16 user clocks independently (the positioning of the clocks in the clock mask are forced by the clock index in `zebu.work/U0.M0/rtb.xref`).
- \$zClockSkewSel and \$zClockFilterSel default value is driverClk. It is recommended to keep this clock reference.
- \$zClockSkewMode default value is pulse.
- \$zClockFilterMode default value is level.

Detailed information about clock modeling is available in the *[ZeBu Compilation Manual](#)* (§3.5 and §5.2 in Rev. C).

3.4.6 Frequency synthesizers for prototyping

When frequency synthesizers are declared in the DVE file as instantiations of the `zClockPort` macro (see Section 2.4.3), their actual frequency can be modified at runtime in the `designFeatures` file, as long as it is lower than the maximum frequency declared in the DVE file (<my_max_freq>, as described in Section 2.4.3).

The appropriate syntax in the `designFeatures` file is:

```
$<my_synthClk>.frequency = <value>;
```

Where <my_synthClk> is the name of the synthesized clock signal and <value> in the runtime frequency that will be used at runtime (float value in kHz).

Example:

If the maximum frequency declared in the DVE file for `my_synthClk1` is 12 MHz, the frequency can be set to 11.5 MHz at runtime:

```
$mySynthClk_1.frequency = 11500;
```



3.5 Declarations for transactors

3.5.1 Global transactor settings

The communication infrastructure for transaction based verification mode uses a burst mode which is applicable by default for all the ports of all the transactors.

The following declaration can be used to modify the settings for the burst mode:

```
$xTor.burstMode = "high | medium | disable";
```

By default, the burst mode is activated (equivalent to "high" value) if the configuration of the PC supports it.

If the PC does not support this communication mode, the ZeBu runtime software (**zServer**) automatically disables the burst mode (equivalent to "disable" value). In such a case, it may be interesting to try the "medium" value which may improve the performance communication when the messages are large. In case of small-size messages, keeping the "disable" value is recommended.

3.5.2 Specific transactor settings

The performance of transactor ports do not only depend on the port sizes, as some big ports may be used very few times while some small ports are used very often. That is the reason why the size of the buffer used to receive messages from the HW can be modified with the number of messages which can fit in this buffer.

Performance is really optimized when an often used port has several messages in its buffer. The current `$<driverName>::<portName>.nbMessMax` is dumped in the `designFeatures.<PCname>.help` file.

The `nbMessMax` setting is specific to a transactor port. Use this setting to declare the maximum number of messages to be sent according to the transactor's memory size:

```
$<driverName>::<portName>.nbMessMax = <value>;
```

3.6 Declaration for Memory Initialization

A memory initialization file can be declared in the `designFeatures` file. This is applicable to any memory instance, whatever its physical implementation (LUTs, registers, BRAM or zrm memory). Detailed information on memory modeling is available in the *[ZeBu Compilation Manual](#)* (§3.8 in Rev. C).

Specification of the file to initialize the external or/and internal memories; the load will occur in the `init()` function.

```
$memoryInitDB = "path_to_memory_init_file";
```

The path can be absolute or relative to the current directory.

Note: this is the only way to initialize BRAM memories when using only **zRun**, since BRAMs are read-only in the **zRun** interface.

Example:

```
$memoryInitDB = "~/mydesign/init_mem1.mem";
```

3.7 Declaration of system clock frequencies

The syntax to modify the frequency of the ZeBu system clocks listed in Table 4 is the following one:

```
$<clock_name>.Frequency = <freq_value>;
```

Where <clock_name> is one of the clocks listed in Table 4 and <freq_value> is the expected runtime frequency, given as an integer (in kHz).

Table 4: Allowed system clock frequencies for runtime

	<clock_name>	Default runtime frequency	DVE parameter
Driver Clock	driverClk	30 MHz or compilation value*	driverfrequency
Master Clock	masterClk	100 MHz or compilation value	masterfrequency
SRAM Trace Clock	traceClk	233 MHz	
Inter-FPGA Communication Clock	xClk	Compilation value	xclkfrequency

* By default, the driverClk frequency is estimated during static timing analysis and automatically set for runtime. However, if the estimated value causes runtime issues, it can be modified by user through the designFeatures file.

Example:

```
$driverClk.Frequency = 20000;
```

3.8 Declaration for driverClk reset signal

A programmable Reset signal driven by driverClk is available if needed for transaction-based verification (it is defined as a number of driverClock cycles). It should only be set when a transactor BFM requires multi-cycle assertions of ureset.

```
$driverClk.Reset = <value>;
```

Where <value> is the number of driverClk cycles while the Reset signal is active.

- Default <value>: 0
- Recommended: unset (comment line with #)

Note: For transactors developed with ZEMI-3, it is necessary to set a non-zero length:

```
$driverClk.Reset = 1;
```

3.9 Declarations for DUT Reset

You can define a reference clock that will be used as a reference for the programming of the DUT Reset settings (the active level of the DUT Reset is declared in the DVE file in the zceiClockPort instantiation):

```
$tgClk.ResetInactive = <nb_inactive_cycles>;
$tgClk.ResetActive = <nb_active_cycles>;
$tgClk.SelectResetClkName = "U0.M0.<my_clock>;"
```



3.10 Declaration for Direct ICE

There are no runtime declarations for Direct ICE. Although they may appear in a commented form in the `designFeatures.<PCname>.help` file, the following parameters come from compilation (`rtb.xref` file) and cannot be modified in the `designFeatures` file:

- `$directIce.GCLK`
- `$directIce.vccIo`
- `$directIce.clock`

3.11 Declaration for Smart Z-ICE

In case of relocation for a design using the Smart Z-ICE interface, the following line should be added in the `designFeatures` file:

```
$smartZICE.connectorRemap_<i> = <j>;
```

Where:

- `<i>` is the compilation index of the Smart Z-ICE connector
- `<j>` is the index of the actual Smart Z-ICE connector at runtime

Note: only the connectors of the unit connected to the host PC can be used for the Smart Z-ICE interface.

Example: When the design was compiled for connectors 0 and 1 but the connectors actually used are connectors 2 and 3, add the following in the `designFeatures` file:

```
$smartZICE.connectorRemap_0 = 2;  
$smartZICE.connectorRemap_1 = 3;
```

However, some other declarations appear as comments in the `designFeatures.<PCname>.help` file but they are automatically set from the compilation and should not be modified in an actual `designFeatures` file:

- `$smartZIce.data`
- `$smartZIce.vcc`
- `$smartZIce.clock`

3.12 Declaration for SVAs

Although it may appear in the `designFeatures.<PCname>.help` file, the `$svaClk` parameter is automatically set from a compilation cross-reference file. It should NOT be modified in the `designFeatures` file.

The only use of this parameter is the declaration of `svaClock` when **zRun** is used without any C or C++ testbench which would initialize the SVAs.

4 Runtime Clock file

4.1 Description

The clock file is an optional input for the Zebu runtime software. It is an alternative to the explicit description of DUT clocks in the `designFeatures` file, and uses the same parameters as the declaration of clocks in the `designFeatures` file, described in Section 3.4.2. There are no specific constraints on the name of the clock file.

4.2 Clock file content

The parameters for clock description in a clock file are the same ones as in the explicit description in the `designFeatures` file (see Section 3.4.2).

4.3 General syntax

The clock file is a text file that uses the following BNF description:

```

file_txt :
    line_txt
    | file_txt line_txt

line_txt :
    '\n'
    | mode '\n'
    | waveform '\n'
    | virtual_frequency '\n'
    | frequency '\n'
    | group_name '\n'
    | Tolerance '\n'

mode :
    '$'CLOCK_NAME.mode '=' 'mode_type' ';'

waveform :
    '$'CLOCK_NAME.Waveform '=' 'WAVEFORMSET' ';'

virtual_frequency :
    '$'CLOCK_NAME.VirtualFrequency '=' NUMBER ';'

frequency :
    '$'CLOCK_NAME.Frequency '=' NUMBER ';'

group_name :
    '$'CLOCK_NAME.GroupName '=' 'STRING' ';'

mode_type :
    controlled
    | free-running
    | in-situ

Tolerance_declaration :
    '$'GroupName.Tolerance '=' 'tolerance_specifier' ';'
    | '$'CLOCK_NAME.Tolerance '=' 'tolerance_specifier' ';'

tolerance_specifier :
    yes
    | no
  
```

Where:

- Terminal token NUMBER has the form: `[0-9]+`
- Terminal token WAVEFORMSET has the form: `[_-]+`
- Terminal token CLOCK_NAME is the name of the clock in the DVE file

5 Memory content file

5.1 Introduction

Memory content files are human-readable text files which describe all or part of a memory. They are used for memory initialization at runtime and for uploading and downloading operations, for any memory instance, whatever its physical implementation (LUTs, registers, BRAM, or zrm memory).

Uploading and downloading memories with memory content files is possible with **zRun** or from a co-simulation testbench.

Memory initialization and uploading/downloading operations can also be done with a ZeBu-proprietary binary format for faster operations. The ZeBu-proprietary binary format for memory content files is not intended for direct reading. It can only be read through a dedicated interface, **hex2bin**.

5.2 Description

The following rules apply to the memory content file:

- Each line in the memory content file starts with the address or the address range to be initialized:

```
| @<address> : <data>
```

or

```
| @<start_addr>,<stop_addr> : <data>
```

The column to separate address values and <data> is optional.

The coma to separate <start_addr> and <stop_addr> is mandatory.

- By default the values for <address> and <data> are given in hexadecimal format ('h, 'H or 0x before the value), but they can also be given in binary ('b or 'B before the value) or decimal ('d or 'D before the value) format.

The default format for the future <data> values can be changed by adding only the format mark on a line, as in the following example (switch to decimal mode):

```
| 'd
```

The default format for the future <address> values can be changed by adding @ with the format mark on a line, as in the following example (switch to decimal mode):

```
| @ 'd
```

- The maximum address size is 32 bits.
- The maximum data size is 4,096 hexadecimal digits. If data are longer than 4,096 digits, only the 4,096 most significant digits (left digits) are considered when loading the memory; other are ignored.

Initializing data in decimal is possible for 32-bit max data.



- When no address is specified, the value corresponds to the address following the previously initialized address. If no explicit address is given at the beginning of the memory content file, the data are written starting from address 0.
- In range mode, only one data can be set on each line of the file.
- If an address is given more than once, the last value for this address is actually used (this can be used, for example, to initialize a memory with a value then to specify values for specific addresses).
- X, x, Z and z values are accepted for <data>; they are changed into 0 in the memory.
- Several data can be listed in the same line of the memory content file, separated by blanks or tabs.
- Underscore characters ("_") can be used inside the data word to improve legibility of initialization words (for example 12345678 can be written 1234_5678).
- C-like line comments start with "//" and end with a line break.

5.3 Example

The following file can be used to initialize a 128 byte memory instance (16-bit words):

```
// Initialize all the memory with value 0x44
@0, FFFF : 44

// Load address 0 with value 0xBA00
@0 : ba00

// Load address 2 with 0xBB02
@2 : BB02
//Load address 3 with 0xCC03 and address 4 with 0xDD04
CC03
DD04

// Switch to binary mode by default for data
'd

// Load address 5 with 0x1234 in binary with legibility separator
@5: 16'b0001_0010_0011_0100

// Load address 6 with a decimal value (will be 0x10 in the memory)
16

//Switch to hexadecimal by default
'h

//Load address 7 with some undetermined bits (z)
@7 0x01z7

// Load range from address FF00 to FF03 with value 0xA
@FF00,FF03: A
```



The actual content of a memory initialized with this example file is following:

Address	Value
0000	BA00
0001	0044
0002	BB02
0003	CC03
0004	DD04
0005	1234
0006	0010
0007	0107
...	...
FEFF	0044
FF00	000A
FF01	000A
FF02	000A
FF03	000A
FF04	0044
...	...

5.4 Memory initialization in the Verilog Source file

When a memory is initialized in the Verilog source file with a `readmemb` or `readmemh` in an `initial` block, **zFAST** automatically generates the memory initialization file for runtime when the following command is added in **zCui**, via the **Additional zFAST Command File** option:

```
compile:rirm=true
```

The resulting file is `zcui.work/design/synth_<rtl_group_name>/readmem.dump`.

This file can be declared in the `designfeatures` file:

```
$memoryInitDB= zcui.work/design/synth_<rtl_group_name>/readmem.dump
```



6 PARFF Parameter File

When the PARFF feature is selected in **zCui** to launch FPGA Place & Route with different sets of parameters, the default parameter set is described in a dedicated file (`fpgaparameterset.xcui`) which is stored in the `$ZEBU_ROOT/etc/ise` directory. The PARFF feature is described in detail in the *[ZeBu Compilation Manual](#)* (§5.4.3 in Rev. C).

Table 5: FPGA P&R parameter sets in the PARFF parameter file

Task Id	Corresponding ISE parameters
L0	MAP_OPTIONS="-ol high -t 2"
L1	MAP_OPTIONS="-ol high -t 3"
L2	MAP_OPTIONS="-ol high -t 4"
L3	MAP_OPTIONS="-ol high -t 5"
L4	MAP_OPTIONS="-ol high -t 6"
L5	XIL_MAP_FULLPACK="1"
L6	XIL_PAR_ENABLE_LEGALIZER="1"
L7	UAP_CLIQUE_LIMIT="10" RT_NO_PLAYOPT="1"
L8	UAP_CLIQUE_LIMIT="10" RT_NO_PLAYOPT="1" XIL_PAR_ENABLE_LEGALIZER="1" XIL_PAR_CLKPH2_MAXIMIZE="1" XIL_MAP_FULLPACK="1"

The PARFF parameter file is an XML file which can be read with a standard text editor. The default PARFF parameter file is given as an example in Section 6.2.

6.1 Modification of the PARFF parameter file

If you are still experiencing failing FPGA compilations with this default parameter sets, you can change the parameters for the PARFF feature by modifying some options for FPGA compilation. For that purpose, you can declare your own PARFF parameter file in **zCui** in the **Superseded ParameterSet file** field (**Advanced** → **Parallel Automatic Recompile of Failing FPGAs** frame).

The default parameter file which is in `$ZEBU_ROOT/etc/ise` directory can be copied for modification to another directory in the user environment, for example with the **zCui** project file (`.zpf` file). This modified parameter file will be declared for the next compilation and the FPGA compilation will be done with specific environment variables to configure FPGA compilation.



6.1.1 Modifying FPGA compilation parameters

All the parameters declared in the PARFF parameter file correspond to modifying environment variables for FPGA compilation tools. Each environment variable is declared in a <parameter> element:

```
<parameter name="env_var">value</parameter>
```

The parameters launched for one task are gathered in a <launch label="name"> tag which name is a free text. The first <launch> tag listed in the parameter file is shown as L0 in **zCui**, then L1 for the second <launch> tag and so on for as many <launch> tags as present in the PARFF parameter file.

Note that the user-defined name of the <launch> tag is visible in **zCui** in the contextual menu for the **Controller** task of each FPGA (**Show Current Parameter Sets** item) in the **Compilation** view.

The following tables show the most frequent options that can be modified by user:

Table 6: Options for ISE Place and Route

NGDBUILD_OPTIONS	Options for ngdbuild
PAR_OPTIONS	Options for par
MAP_OPTIONS	Options for map
BITGEN_OPTIONS	Options for bitgen

Table 7: Other Options for FPGA Compilation

ZEBU_CLEAN_CLOCKS	Optimizes the compilation of clock signals in the FPGAs.
ZEBU_SPLIT_FANOUT	Improves routability for FPGAs with high fan-out nets (fan-out > 256).
ZEBU_SPLIT_FANOUT_OPTIONS	Additional options when ZEBU_SPLIT_FANOUT is set.
ZEBU_LOCK_SOCKET	Adds localization constraints in the FPGAs for communication sockets.
ZEBU_FIX_FLEXPROBE_TRACER	Provides better routability for FPGAs with many flexible probes.
ZEBU_REMOVE_ALL_MUXCY_XORCY	Improves the routability for FPGAs with many MUXCY/XORCY and few LUTs..



6.1.2 Modifying PARFF configuration

In the PARFF parameter file, some options are applicable to the PARFF feature as a whole:

```
<max_depth_of_retry>integer_value</max_depth_of_retry>  
<max_depth_of_relaunch>integer_value</max_depth_of_relaunch>  
<polling_time>integer_value</polling_time>
```

Where:

- <max_depth_of_retry>: depth level for compilation retries. Default is 2.
- <max_depth_of_relaunch>: depth level for compilation relaunches. Default is 1.
- <polling_time>: time interval (in minutes) during which the FPGA status will be updated. Default is 1 minute.

You should refer to the *ZeBu Compilation Manual* (§5.4.3 in Rev. C) for details about these options.

6.1.3 Reserved parameters

The <script> and <retry_script> contents (and the corresponding scripts) are not intended for modification by user. They are reserved to EVE for future improvement of the PARFF feature.

6.2 Example: default PARFF parameter file

The default PARFF parameter file available in \$ZEBU_ROOT/etc/ise contains the following information:

```
<?xml version="1.0" encoding="UTF-8"?>  
<ZeBuUiDoc type="xcui" version="1.0" creator="zCuiFpgaParameterSet">  
<sets>  
  <set label="param1">  
    <script>$ZEBU_ROOT/etc/ise/param1.sh</script>  
    <launch label="t2">  
      <parameter name="MAP_OPTIONS">-ol high -t 2</parameter>  
    </launch>  
    <launch label="t3">  
      <parameter name="MAP_OPTIONS">-ol high -t 3</parameter>  
    </launch>  
    <launch label="t4">  
      <parameter name="MAP_OPTIONS">-ol high -t 4</parameter>  
    </launch>  
    <launch label="t5">  
      <parameter name="MAP_OPTIONS">-ol high -t 5</parameter>  
    </launch>  
    <launch label="t6">  
      <parameter name="MAP_OPTIONS">-ol high -t 6</parameter>  
    </launch>  
    <launch label="fullpack">  
      <parameter name="XIL_MAP_FULLPACK">1</parameter>  
    </launch>
```



```
<launch label="legalizer">
  <parameter name="XIL_PAR_ENABLE_LEGALIZER">1</parameter>
</launch>
<launch label="clique_play">
  <parameter name="UAP_CLIQUE_LIMIT">10</parameter>
  <parameter name="RT_NO_PLAYOPT">1</parameter>
</launch>
<launch label="all">
  <parameter name="UAP_CLIQUE_LIMIT">10</parameter>
  <parameter name="RT_NO_PLAYOPT">1</parameter>
  <parameter name="XIL_PAR_ENABLE_LEGALIZER">1</parameter>
  <parameter name="XIL_PAR_CLKPH2_MAXIMIZE">1</parameter>
  <parameter name="XIL_MAP_FULLPACK">1</parameter>
</launch>
</set>
</sets>
<retry_script>$ZEBU_ROOT/etc/ise/zCpuLoad</retry_script>
<max_depth_of_retry>2</max_depth_of_retry>
<max_depth_of_relaunch>1</max_depth_of_relaunch>
<polling_time>1</polling_time>
</ZeBuUiDoc>
```

7 ZeBu-Server Documents

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

The following Manuals constitute the ZeBu documentation package (some are generic manuals for the ZeBu range):

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



8 EVE Contacts

For product support, contact: support@eve-team.com.

For general information, visit our company web-site: <http://www.eve-team.com>

Europe Headquarters	EVE SA 2-bis, Voie La Cardon Parc Gutenberg, Bâtiment B 91120 Palaiseau FRANCE Tel: +33-1-64 53 27 30
US Headquarters	EVE USA, Inc. 2290 N. First Street, Suite 304 San Jose, CA 95054 USA Tel: 1-888-7EveUSA (+1-888-738-3872)
Japan Headquarters	Nihon EVE KK KAKiYA Building 4F 2-7-17, Shin-Yokohama Kohoku-ku, Yokohama-shi, Kanagawa 222-0033 JAPAN Tel: +81-45-470-7811
Korea Headquarters	EVE Korea, Inc. 804 Kofomo Tower, 16-3, Sunae-Dong, Bundang-Gu, Sungnam City, Kyunggi-Do, 463-825, KOREA Tel: +82-31-719-8115
India Headquarters	EVE Design Automation Pvt. Ltd. #143, First Floor, Raheja Arcade, 80 Ft. Road, 5th Block, Koramangala Bangalore - 560 095 Karnataka INDIA Tel: +91-80-41460680/30202343
Taiwan Headquarters	EVE-Taiwan Branch 5F-2, No. 229, Fuxing 2nd Rd. Zhubei City, Hsinchu County 30271 TAIWAN (R.O.C.) Tel: +886-(3)-657-7626