SystemVerilog Assertions (SVA) supported in ZeBu

AN028
Revision b

ZeBu Application Note

June 2011

Purpose: This document presents how to use SystemVerilog Assertions in the ZeBu environment.

Applicability: This document is applicable for ZeBu software version V6_3_1.

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

Doc Rev.	Product Version	Date	Evolution
b	V6_3_1	June 11	New sections: - SystemVerilog SVA constructs supported in ZeBu, in **Report Only Failure** mode (§2.2.1). - Clock stopping on assertion failure (§3.3.2.1). - Interface of the OnStop callback (§3.3.2.2). - Maximum number of failures report (§3.3.2.3). - Maximum number of assertions in the design in **Report Only Failure** mode (§4.5.1). Updated sections: - SystemVerilog SVA constructs supported in ZeBu, in full report mode (§0). - Screen captures with default selection of **Report only Failure** checkbox (§2.1). - SVA flags described in §2.1 (Synthesis options); in Rev. A, they were described in §2.2 (Backend compilation options). - Log file of compilation flow (§2.2.1). - Message Reporting (§3.1.2). - Changed description of **Live processing** mode and **Clock selector** button in **zRun System Verilog Assertion** panel (§3.2).
a	V4_3_3B	Mar 09	First edition.

Copyright Notice
Proprietary Information

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

Tables

1 Introduction

When synthesizing with **zFAST**, ZeBu supports SystemVerilog Assertions (SVA) for compilation and runtime.

The compilation options are available through **zCui**, the ZeBu compilation interface:

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

At runtime, the assertions which have been synthesized can be processed through the C/C++ API or through **zRun**:

- Selection of assertions.
- Activation of the trigger mechanism on a per-assertion basis.
- Selection of on-line or off-line processing

SVA is a powerful subset of the IEEE 1800™-2005 SystemVerilog standard. The constructs of the SVA subset which are supported in ZeBu are listed in Section 2.2 of the present application note.

The support of SVA with ZeBu requires a specific license feature which can be purchased from your EVE representative and which has to be added as described in the ***ZeBu Installation Manual***.

2 Compilation Flow

The options to compile your design for ZeBu with SVA are available in **`zCui`**.

2.1 Synthesis options

The following **System Verilog Assertions** pane is available in the **SVA** tab of **`zCui`**. Note that this tab is only visible when the **`zFAST`** synthesizer has been selected (**RTL Group Properties → Main** tab → **Synthesizer → zFAST** or **zFAST Script Mode**).

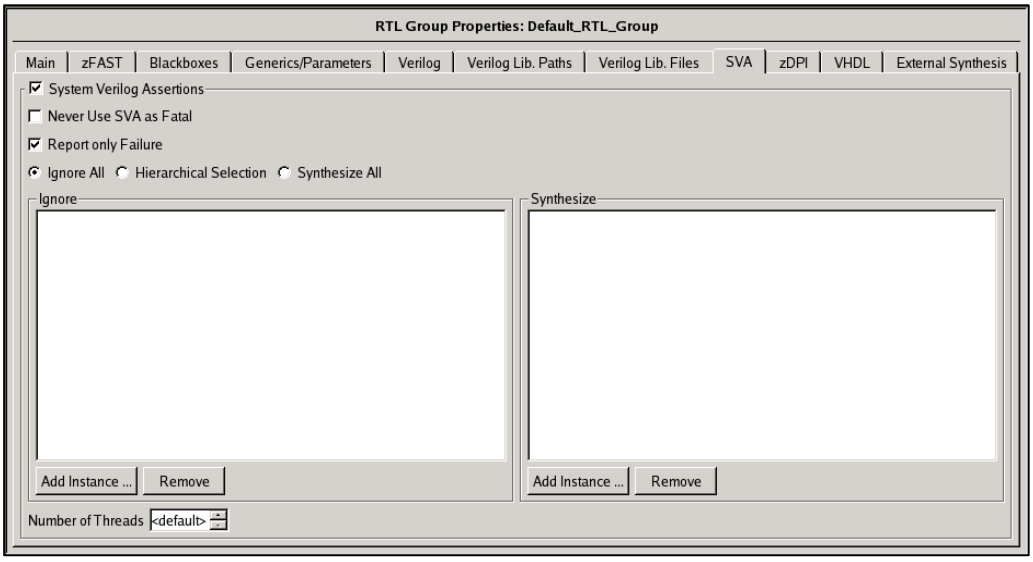

Figure 1: *SVA* tab in `zCui` when the selected synthesizer is *zFAST*

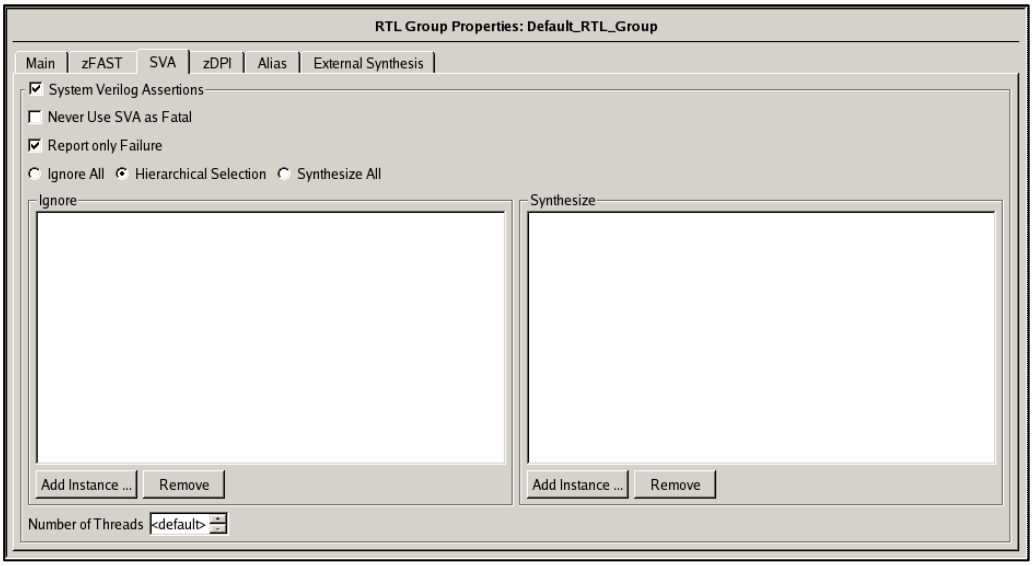

Figure 2: *SVA* tab in `zCui` when the selected synthesizer is *zFAST Script Mode*

If your RTL source files include some assertions, you must select the **System Verilog Assertions** checkbox so that **zFAST** can parse the assertions. In this pane, you can define which assertions will be considered during synthesis, globally (**Ignore All** and **Synthesize All**) or with filtering capabilities (**Ignore** and **Synthesize**).

For filtering capabilities, select the **Hierarchical Selection** checkbox and add the corresponding instances in the **Ignore/Synthesize** frame, with the following rules:

- When an instance is added in the **Synthesize** frame, all assertions under its hierarchy level are synthesized (except those declared in the **Ignore** frame).
- When an instance is added in the **Ignore** frame, all assertions under its hierarchy level are ignored (except those declared in the **Synthesize** frame).
- If no instance is added in the **Ignore** or **Synthesize** frame, all assertions of the design will be synthesized by **zFAST**.

The **Number of Threads** list shows the maximum duration of an assertion, as a number of clock cycles, since one thread is launched at each cycle of the assertion. If an assertion has more cycles than this value, **zFAST** generates a warning and limits the number of created threads for this assertion.

The **Never use SVA as Fatal** and **Report only Failure** checkboxes provide the settings for compilation, which define the default runtime behavior.

By default:

- **Never use SVA as Fatal** checkbox is cleared:
  the SVA trigger is inserted during compilation and allows to stop the emulation on any type of assertion (see Section 3.1.1 for runtime usage).
- **Report only Failure** checkbox is selected:
  only failure times are reported (see Section 3.1.2 for runtime usage).

Note: Make sure that the **Enable BRAM read & write / Write Registers / Save and Restore** checkbox is selected in the **Debugging** panel.

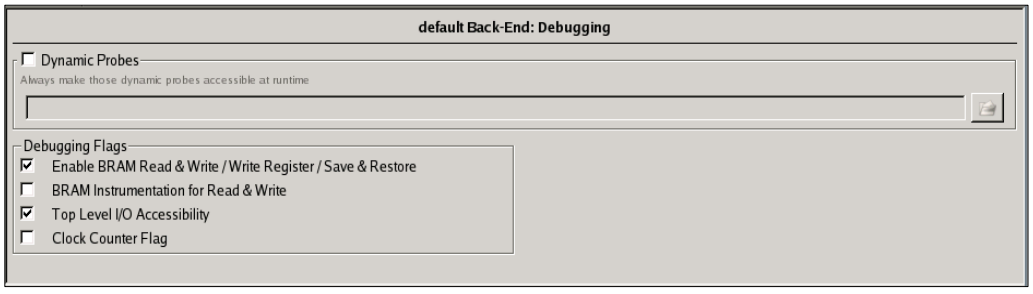

Figure 3: ***Enable BRAM read & write / Write Registers / Save and Restore*** **option**

Before starting emulation, the settings can be modified either in **zRun** or through the C/C++ API.

2.2 Supported SystemVerilog SVA subset

SystemVerilog Assertions (SVA) is a powerful subset of the IEEE 1800™-2005 SystemVerilog standard. The constructs of the SVA subset which are supported in ZeBu are summarized in this section. The list of supported constructs depends on the current setting of the **Report only Failure** checkbox in **`zCui`**:

- **Report Only Failure** mode (default mode in **`zCui`**):
  Current mode when **Report only Failure** checkbox is selected in **`zCui`**.
- Full report mode (start times of SVAs are reported only in this mode):
  Current mode when **Report only Failure** checkbox is NOT selected in **`zCui`**.

Note that the subset supported in "Report Only Failure" mode is smaller than that supported in "Full Report" mode, as stated in Table 1: Supported SystemVerilog SVA subset in "Full Report" mode and Table 2: Supported SystemVerilog SVA subset in "Report only Failure" mode.

The SVA features that were introduced in the SystemVerilog 2009 standard can be enabled by setting the following attribute in the **Additional zFAST Attribute File**:

```
hcsrc:sv2k9=true
```

2.2.1 Supported SVA subset in "Full Report" mode

In the table below, each construct has one of the following statuses:

- **S** in green background: Supported.
- **S*** in yellow background: Supported with the limitations listed after Table 1.
- **U** in red background: Unsupported.

Table 1: Supported SystemVerilog SVA subset in "Full Report" mode

Supported SystemVerilog SVA subset in "Full Report" mode			Status
SEQUENCES	`cycle_delay_range`	`## integral_number`	S
		`##(const_expr)`	S
		`##(const_expr : const_expr)`	S
		`##(const_expr : $)`	S*
	Consecutive repetition	`[* const_expr]`	S
		`[* const_expr : const_expr]`	S
		`[* const_expr : $]`	S*
	Non-consecutive repetition	`[= const_expr]`	S*
		`[= const_expr : const_expr]`	S*
		`[= const_expr : $]`	S*
	Goto repetition	`[-> const_expr]`	S*
		`[-> const_expr : const_expr]`	S*
		`[-> const_expr : $]`	S*
	Sequence operators	`throughout`	S
		`intersect`	S
		`within`	S
		`and` • `and` *of unbounded assertions not supported*	S
		`or` • `or` *of unbounded assertions supported only on left side of implication*	S
		`first_match`	S*
		`.ended`	U

Supported SystemVerilog SVA subset in "Full Report" mode			Status
SEQUENCES (cont'd)	Sampled value functions • *Sampled value function calls outside assertions not supported*	`$sampled`	S
		`$rose`	S
		`$fell`	S
		`$stable`	S
		`$past`	S
		`$changed`	U
	System functions	`$onehot`	S
		`$onehot0`	S
		`$isunknown`	S
		`$countones`	S
	Inferred value functions	`$inferred_clock`	S
		`$inferred_disable`	S
	Global clocking	`$global_clock`	S
	Sequence local variables		S*
	Sequence instantiation		S
	Sequences with formal arguments		S
	Sequences involving function calls		S
	Subroutine call on match of a sequence		S
PROPERTIES	Implicit `first_match`		U
	Property instances	Local instantiations	S
		Cross-module instantiations	U
		Recursive instantiations	U
	`not` • `not` *of property expressions not supported*		S
	`or`		S
	`and`		S
	`if...else`		U
	`disable iff` • *Nested disable not supported*		S
	Implication • *Nested implication not supported*	Non-overlapped	S
		Overlapped	S
	Property instances		S
	Abort properties	`accept_on`	S
		`reject_on`	S
CLOCK DETECTION	Simple clock resolution		S
	Complex clock resolution		S
	Multi-clocked sequences		S*
	Multi-clocked properties		S*
	Clock flow analysis		S
ASSERTION CONTROL TASKS • *Feature supported via C++ API*	`$assertkill`		S*
	`$asserton`		S*
	`$assertoff`		U
CONCURRENT ASSERTIONS	`assert`		S
	`assume`		S
	`cover` • *Supported only for bounded sequences*		S
	Concurrent assertions outside procedural code		S
	Concurrent assertions inside procedural block • *Assertions with clocking event different from clocking event of procedural block not supported*		S*
	Concurrent assertions inside procedural loop		U
DEFERRED ASSERTIONS	• *Supported inside module scope/ always @(clocking event) scope*		S*
IMMEDIATE ASSERTIONS	• *Supported inside module scope/ always @(clocking event) scope*		S*
EXPECT			U

Specific limitations for "Full Report" mode with V6_3_1 software (also listed in the corresponding ***ZeBu-Server Release Note***):

- Sequences:
  - Task/function calls with local variables as arguments are not supported.
  - Among sequence operators, multi-cycle sequences are supported for a maximum of 15 cycles.
  - Nested `first_match` and combination of `first_match` and implication is not supported.
- Properties:
  - Nested implication is not supported.
  - Combination of implication & `first_match` is not supported.
- Clock Detection:
  - Multi-clocked sequences combined with ##0 are not supported.
  - Multi-clocked properties with different leading clocks for sub-properties are not supported.
- Assert Control Tasks:
  - `$assertkill` & `$asserton` are not supported for multi-clocked assertions.
- Concurrent Assertions:
  - Assertions with a clocking event different from the clocking event of the procedural block are not supported
  - Multi-clocked assertions are not supported in procedural block.
- Deferred Assertions:
  - Only supported in module & `always@(clocking event)` scope
- Immediate Assertions:
  - Only supported in module & `always@(clocking event)` scope

2.2.2 Supported SVA subset in *Report Only Failure* mode

In the table below, each construct has one of the following statuses:

- **S** in green background: Supported.
- **S*** in yellow background: Supported with the limitations listed after Table 2.
- **U** in red background: Unsupported.

Table 2: Supported SystemVerilog SVA subset in "Report only Failure" mode

Supported SystemVerilog SVA subset in "Report only Failure" mode			Status
SEQUENCES	`cycle_delay_range`	`## integral_number`	S
		`##(const_expr)`	S
		`##(const_expr : const_expr)`	S
		`##(const_expr : $)`	S*
	Consecutive repetition	`[* const_expr]`	S
		`[* const_expr : const_expr]`	S
		`[* const_expr : $]`	S*
	Non-consecutive repetition	`[= const_expr]`	S*
		`[= const_expr : const_expr]`	S*
		`[= const_expr : $]`	S*
	Goto repetition	`[-> const_expr]`	S*
		`[-> const_expr : const_expr]`	S*
		`[-> const_expr : $]`	S*

Supported SystemVerilog SVA subset in "Report only Failure" mode			Status
SEQUENCES (cont'd)	Sequence operators	`throughout`	S
		`intersect`	U
		`within`	U
		`and` • `and` *of unbounded sequence supported with a maximum duration of 15 cycles*	S*
		`or` • `or` *of unbounded sequence supported with a maximum duration of 15 cycles*	S*
		`first_match`	U
		`.ended`	U
	Sampled value functions • *Sampled value function calls outside assertions not supported*	`$sampled`	S
		`$rose`	S
		`$fell`	S
		`$stable`	S
		`$past`	S
		`$changed`	U
	System functions	`$onehot`	S
		`$onehot0`	S
		`$isunknown`	S
		`$countones`	S
	Inferred value functions	`$inferred_clock`	S
		`$inferred_disable`	S
	Global clocking	`$global_clock`	S
	Sequence local variables		U
	Sequence instantiation		S
	Sequences with formal arguments		S
	Sequences involving function calls		S
	Subroutine call on match of a sequence		U
PROPERTIES	Implicit `first_match`		U
	Property instances	Local instantiations	S
		Cross-module instantiations	U
		Recursive instantiations	U
	`not`		S*
	`or`		S
	`and`		S
	`if...else`		U
	`disable iff` • *Nested disable not supported*		S
	Implication • *Nested implication not supported* • *Combination of implication and* `first_match` *not supported*		S*
	Property instances		S
	Abort properties	`accept_on`	U
		`reject_on`	U
CLOCK DETECTION	Simple clock resolution		S
	Complex clock resolution		S
	Multi-clocked sequences		U
	Multi-clocked properties		U
	Clock flow analysis		U
ASSERTION CONTROL TASKS • *Feature supported via C++ API*	`$assertkill` • *Not supported for multi-clocked assertions*		S
	`$asserton` • *Not supported for multi-clocked assertions*		S
	`$assertoff`		U

Supported SystemVerilog SVA subset in "Report only Failure" mode			Status
CONCURRENT ASSERTIONS	`assert`		S
	`assume`		S
	`cover` • *Supported only for bounded sequences*		S*
	Concurrent assertions outside procedural code		S
	Concurrent assertions inside procedural block • *Assertions with clocking event different from clocking event of procedural block not supported* • *Multi-clocked assertions not supported*		S*
	Concurrent assertions inside procedural loop		U
DEFERRED ASSERTIONS	• *Supported inside module scope/ always @(clocking event) scope*		S*
IMMEDIATE ASSERTIONS	• *Supported inside module scope/ always @(clocking event) scope*		S*
EXPECT			U

Specific limitations for "Report only Failure" mode with V6_3_1 software (also listed in the corresponding ***ZeBu-Server Release Note***):

- Sequences:
  - Task/function calls with local variables as arguments are not supported.
  - Among sequence operators, multi-cycle sequences are supported only on the left side of an implication.
  - `AND` of unbounded assertion is not supported
  - `OR` of unbounded assertions supported only on left side of implication
- Properties:
  - `NOT` of property expressions not supported
- Concurrent Assertions:
  - `cover` supported only for bounded sequences.
  - Assertions with a clocking event different from the clocking event of the procedural block are not supported
- Deferred Assertions:
  - Only supported in module & `always@(clocking event)` scope
- Immediate Assertions:
  - Only supported in module & `always@(clocking event)` scope

2.3 Log file

Two types of files can be found in the synthesis directory:

- `sva_stats_global.txt`: file with global information on SVA synthesis:
  - Total number of SVA assertions present in the design.
  - Total number of LUTs used by SVAs in the design.
  - Total number of Flops used by SVAs in the design.
- `sva_stats/<module_name>.txt` (one file per module):
  These files contain the number of SVAs in the corresponding module, as well as their names and costs in LUT/Flops.

3 Runtime Usage

3.1 Runtime features

At runtime, the assertions which have been synthesized can be processed through the C/C++ API or through **zRun**:

- Selection of assertions.
- Activation of the trigger mechanism on a per-assertion basis.
- Selection of on-line or off-line processing.

Assertions can have 3 states:

- Disabled: no message is generated by this assertion, even if failed conditions are encountered.
- Activated: messages are generated.
- Activated and connected to trigger: messages are generated and the trigger mechanism is enabled for this assertion.

Note that a delay exists between the assertion failure (or success) and the availability of the corresponding message for display.

3.1.1 Trigger mechanism

By default, the ZeBu compilation automatically inserts a trigger (`zsva_trigger`) in the environment so that the assertions can easily be used with the logic analyzer, like any other trigger, to stop the clocks or to start the trace. If **Never use SVA as Fatal** was selected in **zCui**, this trigger is not inserted.

By default, all `$fatal` assertions are connected to this trigger. The `zsva_trigger` trigger will fire whenever a `$fatal` assertion fails at runtime. Any synthesized assertion can be connected to or disconnected from the trigger mechanism at runtime.

3.1.2 Message reporting

When an assertion fails, the following type of message is displayed:

- In full report mode:

```
** <severity>: <message>
Time: 7843 Started 7840 Scope: top.ins0.my_sva File: design.sv Line: 49
```

- In **Report Only Failure** mode:

```
** <Severity>: <message>
Time: 7843 Scope: top.ins0.my_sva File: design.sv Line: 49
```

Where:

- `<severity>:` Assertion severity (`Error`, `Fatal`, `Display`, or `Info`).
  - If the severity is not explicitly set by user (i.e. if the assertion has no error block), the default behavior is `Error` with a standard message.
  - If the assertion has an error block, the error block will be executed. All text messages are supported for all types of error blocks: `$error`, `$fatal`, `$display`, or `$info`.

- `<message>`: type of message:
  - `Assertion error` (when an assertion reports an error).
  - `Assertion passed` (when all assertion messages are displayed).
- `Time`: failure time of the assertion.
- `Started`: start time of the assertion (only in full report mode)
- `Scope`: hierarchical path of the assertion in the design.
- `File`: source file where the assertion is declared.
- `Line`: line number in the source file.

3.1.3 Live Processing vs. Post-processing

At runtime, the messages reported by the SVAs are displayed on screen (`stdout` standard output) and in the runtime log file (with other runtime messages). This operating mode is known as **Live Processing**.

For post-emulation analysis, a report file is dumped with all the messages reported by SVAs. This mode is known as **Post-Processing** and is described in Section 3.4.

3.2 From the zRun interface

The **Global Command** panel now includes a **SVA** button which is active if some SVAs are present in your design and have been synthesized. It opens the **System Verilog Assertion** panel:

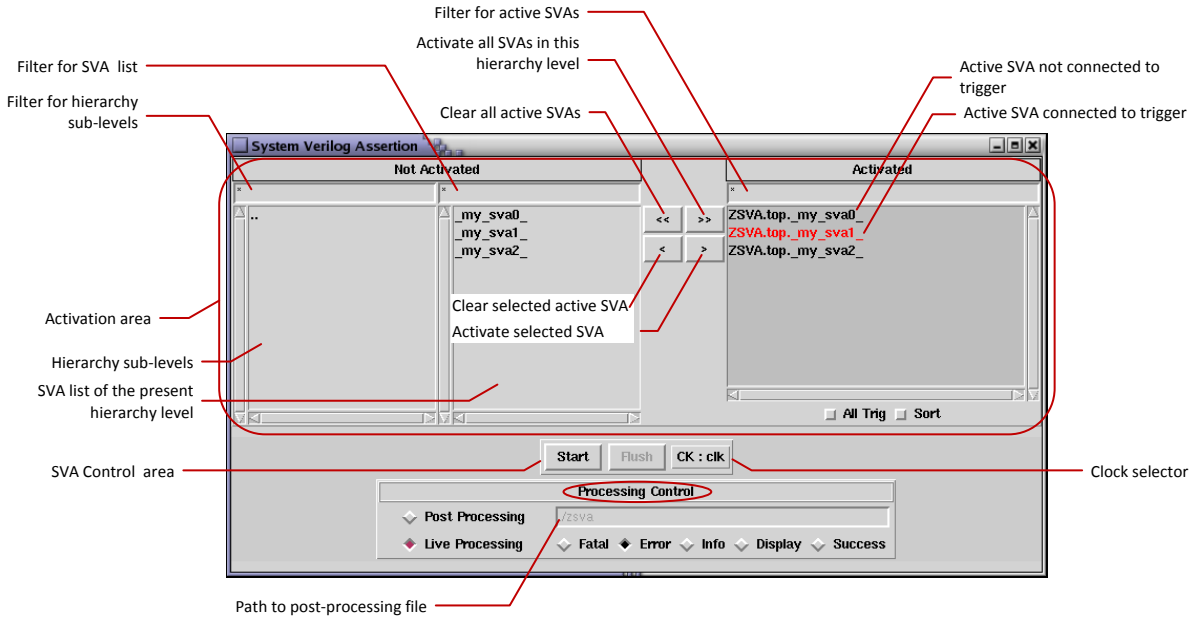

Figure 4: ***System Verilog Assertion*** **panel** ***in*** **zRun**

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**: messages are displayed for the assertions according the selected filter (**Fatal**, **Error**, **Info**, **Display**, or **Success**).
- 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 SVAs 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.
- **Clock selector**: Use this button to select the clock which will be used to generate the timestamps of the reported messages. In the previous figure, the label of the clock selector is **CK : clk** where **clk** is the clock and **CK** its group.

3.3 From a C++ testbench

ZeBu provides an API for both C++ and C interfaces to control SVAs at runtime, for both live processing and post-processing. The SVA class is available in the `$ZEBU_ROOT/include/SVA.hh` header file.

3.3.1 Starting SVA processing

To start the SVA processing the static method `SVA::Start` has to be called. The interface for `SVA::Start` differs according to the type of processing needed. In any case, user needs to declare the following elements:

- The `ZEBU::Board` object
- The name of the clock which will be used for message reporting
- The type of enable can be either:
  - `SVA::ENABLE_REPORT` for enabling messages report only (default)
  - `SVA::ENABLE_TRIGGER` for enabling triggering only
  - `SVA::ENABLE_REPORT | SVA::ENABLE_TRIGGER` for enabling both triggering and report.

The `SVA::Start` method can be called at any time during the run between `Board::open` and `Board::close`.

3.3.1.1 Assertions post-processing

For post-processing, the interface of the `SVA::Start` method includes the name of the report file:

```
static void Start(
          Board *board,
          const char *clockName,
          const char *filename,
          const unsigned int enableTypes = SVA::ENABLE_REPORT
) throw(std::exception);
```

Example:

```
SVA::Start(board,"clk", "dump.zsva", SVA::ENABLE_REPORT);
```

3.3.1.2 Assertions live processing

For live processing, the prototype of the `SVA::Start` method is:

```
static void Start(
          Board *board,
          const char *clockName,
          const unsigned int enableTypes = SVA::ENABLE_REPORT
) throw(std::exception);
```

Example:

```
SVA::Start(board,"clk", SVA::ENABLE_REPORT);
```

3.3.1.3 Specifying a callback at start

A callback with a predefined interface can be used in live processing to handle the action when an assertion is active.

With this callback, it is possible to perform some statistics on the assertions, to write the assertion results in a user file or to format the assertions results in a specific manner.

Prototype of the Start method is slightly different when registering the callback:

```
static void Start(
          Board *board
          const char *clockName,
          ZEBU_SVA_Report callback,
          void *context,
          const unsigned int enableTypes = SVA::ENABLE_REPORT
) throw(std::exception);
```

The parameters passed to this callback are all the necessary data to display the information about the assertion:

Table 3: Parameters for the `ZEBU_SVA_Report` callback

Parameter	Description
`success`	If `success=1`, the call of the callback corresponds to a success of the assertion. If `success=0`, the call of the callback corresponds to a failure of the assertion.
`severity`	This parameter is defined in function of the type of the assertion: the possible values are defined by the `ZEBU_SVA_Severity` type.
`message`	Message to display if user specified one.
`filename`	Name of the file where the assertion is instantiated.
`line`	Line of the SVA instantiation in the source file.
`scope`	Hierarchical path of the assertion in the design.
`startTime`	Start time of the assertion (only in full report mode).
`endTime`	End time of the assertion.
`context`	User pointer.

The interface of the callback can be found in `$ZEBU_ROOT/include/Types.h`:

```
typedef int (*ZEBU_SVA_Report)(
          const int success,
          const unsigned int severity,
          const char *message,
          const char *filename,
          const unsigned int line,
          const char *scope,
          const long long unsigned int startTime,
          const long long unsigned int endTime,
          void *context
);
```

3.3.2 Dealing with assertion failures

3.3.2.1 Clock Stopping on assertion failure

When an assertion fails, the clocks can be stopped instantaneously.
To register an `OnStop` callback and enable Clock Stopping on assertion failure:

```
SVA::EnableClockStoppingOnFailure(Board *board, ZEBU_SVA_OnStop
  callback, void *context)
```

Where:

- `callback`: user callback function.
- `context`: user data pointer to be passed to the callback function when called.

To disable Clock Stopping on assertion failure:

```
SVA::DisableClockStoppingOnFailure(Board *board);
```

3.3.2.2 Interface of the `OnStop` callback

The interface of the `OnStop` callback is the following:

```
typedef int (*ZEBU_SVA_OnStop) (const char **failed_assertion_path,
  const unisgned int *failed_assertion_nb, void *context );
```

Where:

- `failed_assertion_path`: array of `char*` (size: `failed_assertion_nb`). Contains the path of each assertion detected as `failed` during current stop.
- `context`: user data registered pointer.

3.3.2.3 Maximum number of failures report

The following C++ function limits the maximum number of failures to be reported by the SW:

```
SVA::SetMaxFailureNumber(Board *board, unsigned int n)
```

Where:

- `n`: maximum number of failures. When this number has been reached, the assertion is disabled. It is reset to an infinite value by each call to `SVA::Stop`.

The equivalent C function is:

```
ZEBU_SVA_SetMaxFailureNumber(ZEBU_Board *board, unsigned int n)
```

3.3.3 Stopping SVA processing

To stop SVA processing the method `SVA::Stop` has to be called.

Example:

```
SVA::Stop(board);
```

The method `SVA::Start` has to be called before the `SVA::Stop` method and that `SVA::Stop` method has to be called before the `Board::close` method.

3.3.4 Changing the reported severity level

By default only the assertions without action block or with an action block containing a `$error` or `$fatal` display a message on the screen when failing.

The `SVA::SelectReport` method changes the minimum level of severity from which messages are displayed:

```
static void SelectReport(Board *board, const unsigned int severity
= ZEBU_SVA_Failed_Display) throw(std::exception);
```

The available values for severity are defined by the `ZEBU_SVA_Severity` type in the `ZEBU_ROOT/include/Types.h` header file:

```
ZEBU_SVA_Failed_Fatal      (highest)
ZEBU_SVA_Failed_Error
ZEBU_SVA_Failed_Warning
ZEBU_SVA_Failed_Info
ZEBU_SVA_Failed_Display
ZEBU_SVA_Success           (lowest)
```

Example: When the severity argument is set to `ZEBU_SVA_Failed_Warning`, assertions which have an action block with `$warning` and higher severities (`$error` and `$fatal`) are displayed.

3.3.5 Disabling/Enabling assertions and connecting to the SVA trigger

The activation of an assertion can be modified with `SVA::Set` method. By default:

- All the synthesized assertions are active and produce messages.
- Only fatal severity assertions are connected to the trigger `zsva_trigger`.

The prototype is the following:

```
static void Set(
          Board *board,
          const unsigned int types = SVA::ENABLE_REPORT,
          const char *regularExpression = NULL,
          const bool invert = 0,
          const bool ignoreCase = false,
          const char hierarchicalSeparator = '.'
) throw(std::exception);
```

Where the arguments are:

- `types`: initialization value of a SVA or a group of SVAs. Possible values:
  - `SVA::ENABLE_REPORT` for enabling messages report only
  - `SVA::ENABLE_TRIGGER` for enabling triggering only
  - `SVA::ENABLE_REPORT | SVA::ENABLE_TRIGGER` for enabling both triggering and report.
  - `0` for nothing.
- `regularExpression`: path of a SVA or regular expression for a SVA group.
- `invert`: if `true` inverts the selection done by the regular expression.
- `ignoreCase`: ignore case sensitivity for the regular expression.
- `hierarchicalSeparator`: set hierarchical separator for regular expression.

3.4 Post-processing the SVA report file with `zsvaReport`

The tool for post-processing of the SVA data is **`zsvaReport`**. It takes as inputs the SVA report dump file, the path to `zebu.work` and the severity level. It processes the content of the file and displays the assertion messages according to the selected severity level.

The command line for **`zsvaReport`** is the following:

```
$ zsvaReport -i <zsva_filename> [-z <zebu.work>] [-s <severity> ] [-h]
```

Table 4: Options for `zsvaReport` tool

Option	Description
`-i <.zsva filename>`	Name of the SVA report dump file to read.
`-z <zebu.work>`	Path to the SVA report dump file. Default is `./zebu.work`
`[-s <severity> ]`	Minimum severity level for which the messages are displayed. Possible values are: `fatal`, `error`, `info`, `display` and `success`. Default is `error`.
`-h`	Displays on-line help.

Example:

```
$ zsvaReport -i dump.zsva -z zcui.work/zebu.work –s info
```

4 Limitations

4.1 Multiple clocks used by SVA vs. HDL co-simulation

Using the SVA feature with multiple clocks is supported (one single clock per assertion). However in order for SVA to work correctly with multiple clocks, the clocks used by SVA must be grouped in the `designFeatures` file.

A conflict arises when you use HDL co-simulation because multiple clocks must not be grouped in this case (ungrouped by default). If the clocks are not grouped and you click the **SVA** button in the **zRun** GUI, the following error will be generated:

```
-- ZeBu : zRun : ERROR : ZRUN0118E : ZEBU_Sva_getSamplingClock : Clock group name not found
```

The same error will be generated if the `ZEBU_Sva_getSamplingClock` function is used in one of the following cases:

- Directly from the **zRun Tcl command** field:
  ```
  > ZEBU_Sva_getSamplingClock
  ```
- Via a Tcl script executed from the **Tcl command** field with the `-do` option:
  ```
  > zRun -do <my_script.tcl>
  ```

4.2 Reported timestamps

The timestamps reported in the assertions failures are fully deterministic but are not relative to a user-defined clock. They are relative to a deterministic clock called 'SVA clock' which is an implicit clock related to the clock group selected by user.

This clock has a posedge for each clock event in the selected group.

4.3 Simultaneous use of flexible probes

Flexible probes and SVAs can be used in the same design but the sampling clock for Flexible Probes is the SVA clock, also if a different sampling clock was declared when compiling the design. This limitation exists as soon as some assertions are synthesized; it may cause an oversampling for the Flexible Probes.

4.4 Usage of "%" parameter in `$display` system tasks

Using a `$display` which includes a variable does not display the variable but the non-interpreted string which is in the `$display`.

Example:

```
if(cond == true) $display("PASSED : DATA = %d", data);
else $display("FAILED: DATA = %d", data);
```

In case of failure, the message displayed at the screen will be:

```
'FAILED: DATA = %d'
```

4.5 Maximum number of assertions in the design

4.5.1 *Report Only Failure* mode

When the **Report only Failure** checkbox is selected in **`zCui`**, the maximum number of assertions is:

Nb_FPGA * 64 * 480

Example:
In a 64-FPGA ZeBu system, the maximum number of assertions is 1,966,080.

4.5.2 Full report mode

When the **Report only Failure** checkbox is not selected in **`zCui`**, the depth of each SVA has to be taken into account.

The maximum number of SVA threads for the complete ZeBu system is:

Nb_FPGA * 64 * 480 / 2

The sum of depths for all SVAs must not exceed this maximum number of available SVA threads.

If all assertions are immediate assertions (depth = 1), you can have as many assertions as the maximum number of SVA threads.

Example:
In an 8-FPGA ZeBu system, the maximum number of SVA threads is 122,880. As a consequence, the maximum number of assertions is 122,880 if they are all immediate assertions.

4.6 Assertions vs. Multiple RTL groups

The design cannot be synthesized with SVAs if the assertions are split in different RTL groups.

5 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 Room 806 8F, No. 20 GuanChian Road Taipei, Taiwan 100 Tel: +886 2 2375 9275