ZeBu® Customer DPI Based Transactors User Guide

Version V-2024.03-1, July 2024

Feedback

Contents

Preface

This section includes the following topics:

- About This Book
- Contents of This Book
- Related Documentation
- Typographical Conventions
- Synopsys Statement on Inclusivity and Diversity

About This Book

The *ZeBu® Custom DPI Based Transactors User Guide* describes ZEMI-3 that enables writing transactors for functional testing of a design.

Contents of This Book

The *ZeBu® Custom DPI Based Transactors User Guide* has the following chapters:

Chapter	Describes...
ZEMI-3 Transactors: An Overview	Introduces ZEMI-3 transactors and factors that involve in deciding a transactor architecture.
Data Exchange Between Hardware and Software	Factors that affect data exchange between hardware and software
Clocks	The types of clocks that can be used in a ZEMI-3 transactor
Advanced Features	The advanced features supported by the ZEMI-3 infrastructure
Writing the Hardware Part	The steps when writing the hardware part of a ZEMI-3 transactor
Writing the Software Part	The steps when writing the software part of a ZEMI-3 transactor

Chapter	Describes...
Compiling the Hardware Part of a ZEMI-3 Transactor	The steps used for compiling the hardware part of a ZEMI-3 transactor
Running Emulation of ZEMI-3 Transactors	The steps to run hardware emulation of a ZEMI-3 transactor
Running Emulation in a Heterogeneous Environment	The steps to run hardware emulation of a ZEMI-3 transactor in a heterogeneous environment
Performing Simulation of ZEMI-3 Transactors	The steps to perform simulation of the ZEMI-3 transactor

Related Documentation

Document Name	Description
ZeBu User Guide	Provides detailed information on using ZeBu.
ZeBu Debug Guide	Provides information on tools you can use for debugging.
ZeBu Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu UTF Reference Guide	Describes Unified Tcl Format (UTF) commands used with ZeBu.
ZeBu Power Aware Verification User Guide	Describes how to use Power Aware verification in ZeBu environment, from the source files to runtime.
ZeBu Functional Coverage User Guide	Describes collecting functional coverage in emulation.
Simulation Acceleration User Guide	Provides information on how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
ZeBu Runtime Performance Analysis With zTune User Guide	Provides information about runtime emulation performance analysis with zTune.
ZeBu Custom DPI Based Transactors User Guide	Describes ZEMI-3 that enables writing transactors for functional testing of a design.

Document Name	Description
ZeBu LCA Features Guide	Provides a list of Limited Customer Availability (LCA) features available with ZeBu.
ZeBu Synthesis Verification User Guide	Provides a description of zFmCheck.
ZeBu Transactors Compilation Application Note	Provides detailed steps to instantiate and compile a ZeBu transactor.
ZeBu zManualPartitioner Application Note	Describes the zManualPartitioner feature for ZeBu. It is a graphical interface to manually partition a design.
ZeBu Hybrid Emulation Application Note	Provides an overview of the hybrid emulation solution and its components.

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	`<sig-name>`
Message	Active low signal name '<sig-name>' must end with _X.
Message location	`OUT` <= IN;
Reworked example with message removed	`OUT_X` <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry

Syntax	Description
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
... (Horizontal ellipsis)	Other options that you can specify

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

Feedback

1

ZEMI-3 Transactors: An Overview

This section contains the following topics:

Introduction to a ZEMI-3 Transactor

ZEMI-3 is a behavioral SystemVerilog compiler for transactor Bus Functional Models (BFMs). It is compatible with the Standard Co-Emulation Modeling Interface (SCE-MI) 2.0 specifications.

ZEMI-3 simplifies writing cycle-accurate BFMs and exchanging data with a C++ or SystemVerilog testbench. Its infrastructure enables writing transactors for functional testing of a design.

ZEMI-3 transactors are based on the following important concepts:

- The communication between the hardware and software parts uses inter-language function calls, based on the DPI mechanism, which is part of the IEEE standard for SystemVerilog (IEEE 1800-2012).
- The description of the hardware part of the transactor uses behavioral coding style.

Definition of a ZEMI-3 Transactor

A transactor organizes the interaction between the testbench and DUT:

- The software part of the transactor is written in C/C++; it interacts with your testbench through high-level commands.
- The hardware part interacts with the DUT at a cycle-level or signal-level; it processes multicycle operations as an untimed functional model. These interfaces ease the integration of the testbench and DUT.

In a ZEMI-3 transactor, the hardware and the software interact through inter-language function calls. The transaction data correspond to the arguments of the function call, data transfers, and time synchronizations happen automatically. The calls can be initiated by the hardware or software according to the verification environment.

Factors for Choosing a Transactor Architecture

Selecting the transactor architecture depends on the factors:

- Choice of the Hardware and Software Parts
- Bandwidth Required
- Number of Clock Cycles in a Transaction
- Choice of a Modular Transactor Architecture

Choice of the Hardware and Software Parts

A transactor is composed of hardware and software parts and some operations work better in one part or the other. For example, complex arithmetic operations are more efficient in software and require a lot of resources in hardware. On the other hand, bit-level operations such as CRC calculations are more efficient in hardware.

Bandwidth Required

With ZEMI-3, it is simple to stream the data out of the DUT for analysis purposes. However, streaming thousands of bits at every cycle is not an efficient way of using ZeBu, and may result in performance degradation.

When data is required by the software part, send only the necessary data.

Number of Clock Cycles in a Transaction

ZEMI-3 performance is maximized when multiple cycles are executed for each interaction between the software and hardware parts of the transactor. The more cycles in a single transaction, the faster it runs.

The ratio between the number of clock cycles and the number of inter-language calls should be maximized for the best performance.

Choice of a Modular Transactor Architecture

When implementing a complex protocol such, as PCI Express, it is important to create a modular architecture with separate software-initiated commands, such as, Read PCI and Write PCI. Such a modular architecture makes it easier to write transactor software.

It also makes it easier to upgrade a transactor, that is, to add a new feature, you can create a separate function without impacting the existing code.

Advanced features, such as, protocol mode or serialized calls can help with modular transactor architecture.

Feedback

2

Data Exchange Between Hardware and Software

An exchange can be initiated by the hardware or the software according to your verification environment. The choice for hardware or software -initiated exchanges depends on the architecture of the transactor.

See the following sections in this chapter:

Simulation Time Consumption

The simulation time consumption concept indicates that tasks consume simulation time (that is, clock cycles) when they wait for particular events or clock edges.

When using the SystemVerilog DPI, tasks consume time while functions must return instantaneously, without consuming time.

Hardware -Initiated Transactions

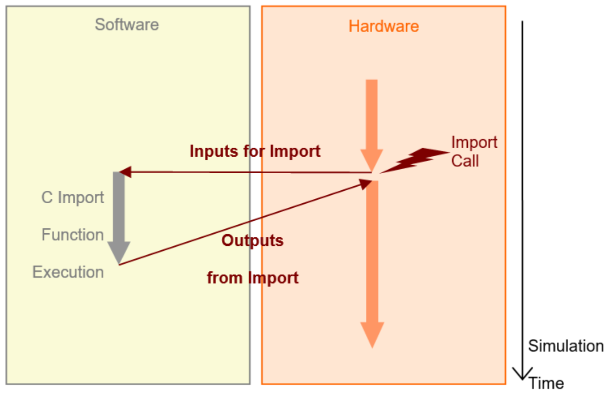
Hardware-initiated transactions are preferred to transfer data from hardware to software. In such cases, the hardware part typically requires a complex operation result, which is easier to implement in the software part.

You can use an import function, which is a C-language function called from the SystemVerilog code. This function has optional inputs that send data to the software and

Feedback

outputs that return data to the hardware. This mechanism allows the hardware part to send data to the software and then use the results produced by the software part.

Figure 1 Hardware-Initiated Transaction

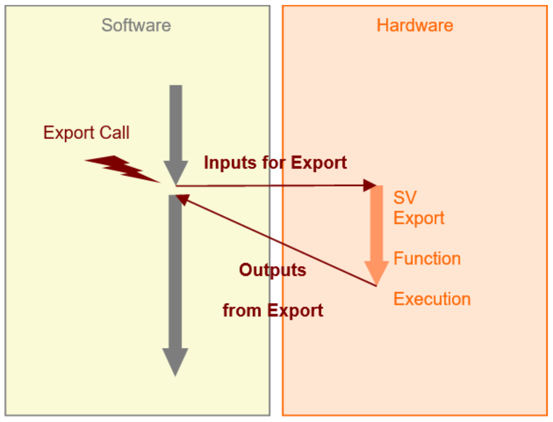


During the execution of an import function (in the software part), the hardware part of the transactor stops (the design clocks do not run). Such calls degrade the performance if used frequently, such as, at every clock cycle. The C import function is executed in its own software thread.

Software -Initiated Transactions

Software-initiated transactions are preferred to transfer data from software to hardware. In such cases, you can use a SystemVerilog export function. The software part of the transactor provides the inputs for this function. It allows the software part to send data to the hardware part and then use the results produced by the hardware.

Figure 2 Software-Initiated Transactions

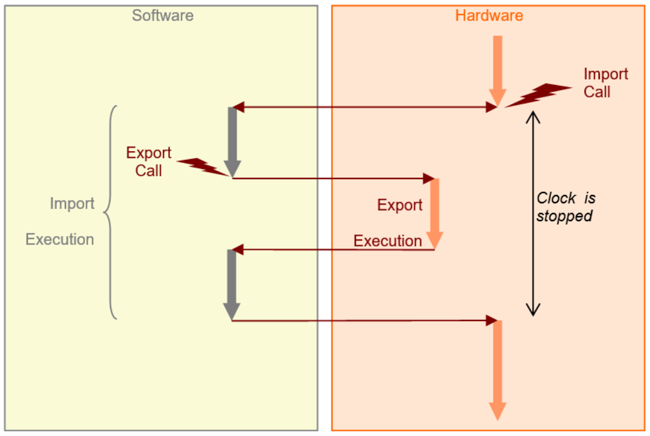


During export function execution, the software part of the transactor waits for the outputs to be returned by the hardware.

Mixing Imports and Exports

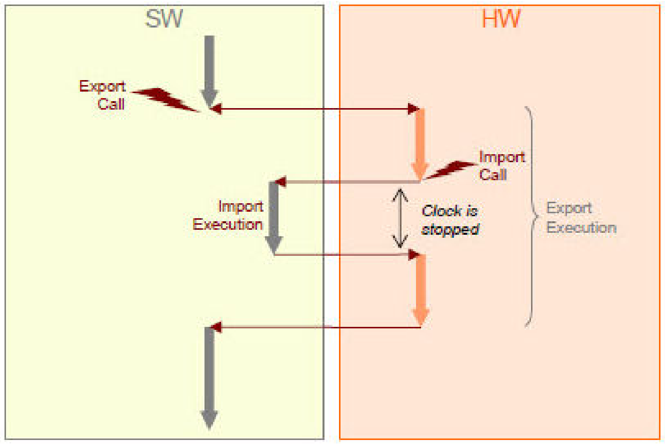
In ZEMI-3, an export function can be called from an import function and an import function can be called from an export function or task. The maximum level of nesting is two, that is, one import can call multiple exports; however, these exports cannot call another import.

Figure 3 Calling an Export From an Import (Context Import)



An import function that calls an export must be declared as a context import, as described in Import Function. The export called from the import must not consume DUT clock cycles, because ZEMI-3 imposes the restriction that import functions must not consume simulation time.

Figure 4 Calling an Import From an Export



The export can consume simulation time. However, the included import cannot consume simulation time.

Functions and Tasks

For import calls only functions can be used because the software part does not directly control simulation time of the hardware part. From a hardware point of view, functions execute immediately, which means that any call of an import function stalls the transactor clock.

For export calls, both tasks and functions can be used but only tasks can wait for events to happen. Functions can only set or read values and must return immediately.

Export tasks and functions are executed by the software calls. They execute in parallel with all other hardware blocks. Only one instance of an export call can be active at any one time. If there are multiple calls, they execute sequentially (the second such export task waits until the first task finishes.)

Clock events can be used directly or through wait statements, as displayed in the following example:

```
always
begin
  @(posedge clk)
  a <= b;
  wait(clk == 1'b0);
  a <= c;
end
```

If a task that has no wait or event statement it should be written as a function.

Function or task arguments are transferred across the hardware and software parts without any delay at the synchronization points, see Synchronization Points. The ZEMI-3 infrastructure prevents the channels from overloading.

Import tasks are not supported in the ZEMI-3 infrastructure, therefore, an import does not consume any simulation time; that is, in case of nesting, only an export function (not an export task) may be called from an import.

The size of the data exchanged between hardware and software at a synchronization point can be up to 8,000 bits in each direction (input or output). If the data is bigger than this limit, the messages are automatically turned into multicycle messages, hence, impacting runtime performance.

Managing the Call Order of Software Import Functions

When functions are called in the same SystemVerilog process (always or initial block), the order of calls is guaranteed to be the same on both hardware and software parts, even in a streaming environment.

When function is called in separate processes, the order in which they execute on the software part can differ from the order in which they were called by the hardware part. This order can be made deterministic through the serializing calls feature, see Serializing Calls.

Synchronization Points

Synchronization points are needed when bi-directional data exchanges occur between the hardware and software parts of the transactor. In this case, the hardware and software parts need to wait for each other.

It is recommended to avoid functions or tasks that mix inputs and outputs to minimize synchronization points. For example,

- Using an export function or task with outputs (or with a return code) prevents the software from performing any other operation in parallel.
- Using a non-streaming import function in a clocked always block stops the clock (for more details about streaming data, see Streaming Data). If required, replace the import function by a streaming import function and a streaming export function.

Such cases should be rare, or should correspond to a large number of cycles of activity in the design.

Synchronization points are required to create the full environment. However, the synchronization points should be minimized to limit the impact on performance. Too many synchronization points may result in frequent communication and degrade performance. The best performance can be achieved when those points are performed after a large number of cycles, or completely avoided as in data streaming.

Streaming Data

It is important to remove the synchronization points in a pipe-like communication channel in which the data moves in a single direction: software to hardware or hardware to software. Such situations exist when import or export functions have only inputs. Functions with only outputs cannot be considered for streaming because they require the following bi-directional exchanges:

- When the call is made; although, there is no input.
- When the outputs are sent back.

Figure 5 Streaming Import

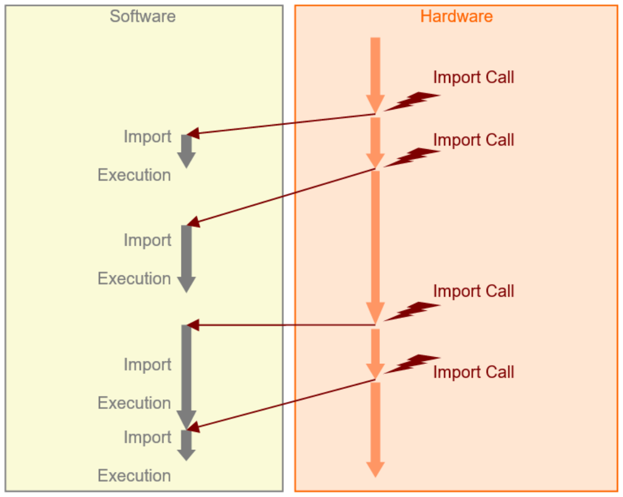
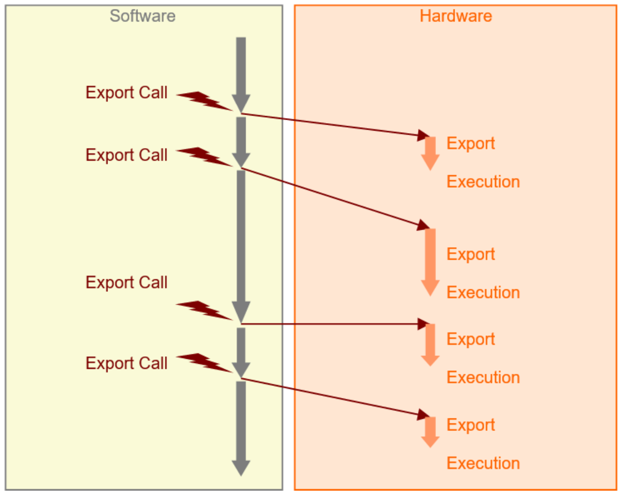


Figure 6 Streaming Export



It is possible for ZEMI-3 to turn function calls into a streaming communication to avoid synchronization points and to optimize the transactor performance. When there are only inputs, hardware and software parts can execute concurrently without having to wait for one another.

In streaming mode, the ZEMI-3 infrastructure automatically optimizes the communication speed by buffering the requests and avoid any possible overloading of the communication channel.

The following call types are automatically processed as streaming communication:

- Non-context import functions with inputs only
- Export functions with inputs only
- Export tasks with inputs only

The automatic conversion to streaming mode can be disabled for some call types. Context import functions can also be forced into streaming mode, see Forcing/Disabling Streaming Mode. However, nested import and export functions (see Mixing Imports and Exports), must never be forced into streaming mode, because it causes unexpected runtime behavior.

It is recommended to create transactions in the streaming mode otherwise ZEMI-3 cannot optimize communication in the presence of synchronization points.

If output-only import functions are used significantly by the environment, the prefetch mechanism may improve performance. For more details, see Prefetch Calls.

In ZEMI-3 transactors, `zceiMessageOutPort` is used to send arguments of import DPI calls from hardware to software.

To stop sending constant arguments of import DPI call through `zceiMessageOutPort`, use `ZEMI-3 -optimize_dpi_constant_args true`.

This can help optimize and save hardware - software communication bandwidth and the hardware capacity by reducing the size of `zceiMessageOutPort`.

3

Clocks

This chapter contains the following subsections:

Controlled Clocks

A ZEMI-3 transactor must have at least one controlled clock. Such clocks are input ports to the transactor and they are declared in the hardware top by instantiating a `zceiClockPort` (see Instantiating a ZEMI-3 Transactor in the Hardware Top).

Controlled clocks are enabled by default. If no data is available from the software or the software is not ready to receive data during a DPI call then all controlled clocks of the transactor are automatically stopped until the data becomes available or the software is ready. If there are multiple clocks, all the groups to which they belong are stopped.

ZEMI-3 transactor supports sample clock optimizations to avoid the sampled clock effect by setting `zemi3 * -sample_clock_mode` as `true`.

Sampled Clocks

In some cases, clocks can be generated by the DUT as in an LCD design. This kind of clock is known as a sampled clock, and it must not be declared in the same way as a controlled clock.

Ensuring the Best Performance

For higher performance it is best to limit the number of clocks used in a transactor. If possible, avoid divided clocks and gated clocks.

Using sampled clocks may slow down the transactor because the controlled clocks have to be stopped to ensure consistent inputs from the design.

From a ZEMI-3 point of view, any signal used in a `@(posedge <signal>)` or `@(negedge <signal>)` statement is considered a controlled or sampled clock.

Feedback

4

Advanced Features

For implementation details of the features described in this ZEMI-3 section, see Supported System Calls.

The following are the advanced features supported by the ZEMI-3 infrastructure:

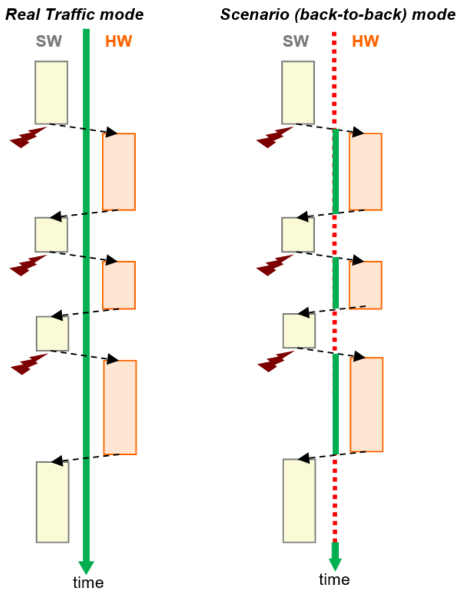
- Back-to-Back Mode
- Protocol Mode
- Prefetch Calls
- Serializing Calls
- Lossy Calls
- Port Buffering
- Multicycle Messaging (Port Multiplexing)
- Automatic Compiler Optimization

Back-to-Back Mode

When verifying your design using transactors, you must consider the following verification strategies:

- System Functional Testing: In Real Traffic mode, transactor clocks are always running. The delay between consecutive transactions is not deterministic.
- Corner-Case Testing: In Scenario mode, insertion of idle cycles between transactions is forbidden except when explicitly requested (clocks have to be controlled in such a way that only requested cycles are processed). This is also called the back-to-back mode.

Figure 7 Back-to-Back Mode



During the verification process, you must switch from the real traffic mode to the scenario mode many times, with a minimum development cost. The back-to-back mode enables you to design your transactor as compliant with your real-traffic specifications. The same transactor can be operated in either mode.

The back-to-back mode can be selected at runtime (from the C/C++ API). The initial back-to-back mode can be set during compilation (in UTF file).

Protocol Mode

When several export functions (or tasks) use shared resources, it is important to prevent concurrent accesses from different processes in order to avoid read/write conflicts.

The ZEMI-3 infrastructure addresses this via the protocol mode, which automates the scheduling of functions (and tasks).

This mode is activated by declaring an additional advanced property when the export functions are declared in the SystemVerilog code.

Streaming mode can be used for protocol-gathered export tasks and functions.

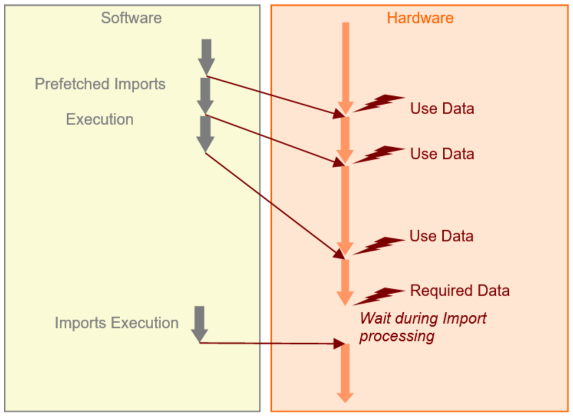
Prefetch Calls

When an import function has only outputs, the software can sometimes prepare data so that the hardware part does not have to wait for the data at the synchronization points. This is known as prefetch calls.

- Without prefetching, an import call causes the hardware to request new data from the software. The software services the request by sending the data back to the hardware. The clocks then restart until the next call.
- Prefetching (for an import with output data only) eliminates the need to wait for the actual call to prepare the data for the (next) request.

Prefetching is similar to the ZEMI-3 streaming mode for software to hardware communication, except that the software controls the communication flow.

Figure 8 Prefetch Mode



The compiler automatically creates a specific prefetch function for each import function with only outputs. The software calls the prefetch function and stores the results until they are needed by the hardware part.

Prefetch mode can be enabled or disabled independently for each:

- When enabled, the original import function is not called; instead, the data is provided by the prefetch function.
- When disabled, the import function is called.

Serializing Calls

When multiple processes call import functions, ZEMI-3 does not guarantee that the software handles the calls in the same order as they were called in the hardware.

When two functions are called at different times, the later call may be executed first by the software.

When it is important to preserve the call order, you should explicitly serialize the calls.

A general guideline is that software functions that operate on the same data must be serialized (for example, read and write calls to the same memory).

Serializing calls allow you to use the same hardware port for several imports. However, if an import requires a high throughput then it should not be serialized.

Lossy Calls

This feature can be specified for an import function to keep it from sending messages to the software when the communication channel is busy. For more details, see Enabling System Calls.

Port Buffering

This feature can be specified for a streaming import function to create buffers for the message. This can be useful when exchanging small packets that can tolerate higher latency. For more details, see Port Buffering.

Multicycle Messaging (Port Multiplexing)

This feature allows splitting messages into smaller packets, which allows for smaller hardware ports. This feature applies to imports and exports and can be set in the UTF file.

Automatic Compiler Optimization

Compiler optimization settings for the hardware part of the transactor can be specified in the UTF file, in particular for hardware-software communication. These settings combine the advanced features described in Advanced Features, according to the environment and constraints.

Feedback

5

Writing the Hardware Part

For more information, see the following subsections:

- Imports and Exports in SystemVerilog Code
- Using Behavioral Coding Style
- Adding Delta Delays
- Supported System Calls
- Advanced Properties

Imports and Exports in SystemVerilog Code

This section consists of the following subsections:

- Import Function
- Export Function/Task
- Blocking Assignment in Export Tasks

Import Function

An import function declared in SystemVerilog can be called by the hardware part of the transactor. Both the declaration and the call reside in the same Verilog module.

The function is implemented in C by the software part of the transactor. For more details, see Implementing Import Functions in C.

For example:

```
// Import declaration in SystemVerilog
import "DPI-C" function void read_addr(input bit [31:0] addr, output bit
 [31:0] data);

// Import call in SystemVerilog
reg addr;
reg data;
always @(posedge clk)
```

```
begin
  addr <= addr + 1;
  read_addr(addr, data);
  val <= data;
end
```

When you want to use scope information, the import function must be declared explicitly as a context import. This is required when the import function is nested within an export function.

For example:

```
import "DPI-C" context function void send_one_addr(input bit [31:0] addr,
 input bit [31:0] data);
```

Export Function/Task

For exports, both the declaration and the implementation of the function (or task) reside in the same SystemVerilog module.

The export declaration does not include the function prototype; the prototype is specified only in the implementation.

For example:

```
// Export declaration in the SystemVerilog code
export "DPI-C" task write_one_addr;

// Export implementation in the SystemVerilog code
task write_one_addr(input bit [31:0] addr, input bit [31:0] val);
  begin
    mem_we <= 1;
    mem_addr <= addr;
    mem_din <= val;
    @(posedge mem_clk);
    mem_we <= 0;
    @(posedge mem_clk);
  end
endtask
```

Blocking Assignment in Export Tasks

Non-blocking assignments in export tasks can result in unexpected runtime behavior, in particular when assigning a value to an output argument as the final statement of the task.

For example:

```
task receiveCmd(input bit[31:0] cmd);
  begin
    @(posedge clk);
    while (!lastCmdEnd)
      @(posedge clk);
    nextCmd = cmd;
    newCmdEn = 1'b1;
    @(posedge clk);
    newCmdEn = 1'b0;
  end
endtask
```

A non-blocking assignment to an output argument never passes the correct value to the software because the non-blocking assignment is not executed correctly. Always use blocking assignments in export tasks to avoid such situations.

Using Behavioral Coding Style

The hardware part of a transactor designed with ZEMI-3 can be written using behavioral code, which normally cannot be synthesized. This section has the following subsections:

- Supported Behavioral Subset
- Clocks, Multiple Clocks, and Multiple Edges
- Usage of Blocking and Non-Blocking Assignments
- Limitations

Supported Behavioral Subset

The following constructs are supported in the behavioral description of the hardware part of a transactor:

Table 1 Supported Sub-Set of Behavioral Code

Supported Constructs	Example
Standard synthesizable RTL subset	-
User-Defined Primitives (UDPs)	Typically, Xilinx primitives
initial statements	`initial counter = 0;`

@edge in always/initial statements	`initial begin`
	`  a = 0;`
	`  @(posedge clk);`
	`  a = 1;`
	`end`

Table 1 Supported Sub-Set of Behavioral Code (Continued)

Supported Constructs	Example

@edge in if and loops	`for (i=0;i<128;i=i+1) begin`
	`  @(posedge clk);`
	`  mem[i] = 0;`
	`end`

Multiple clocks/edges in the same process	`always @(posedge clk) begin`
	`  a = 0;`
	`  @(negedge clk);`
	`  a = 1;`
	`  @(negedge reset);`
	`  a = 2;`
	`end`

Complex edge combination	`always @(posedge clk or negedge clk2)`
Edge and level events	`always @(clk or reset or posedge sig)`

Unbounded loops	`Initial`
	`  while (1) begin`
	`   @(posedge clk1)`
	`    c <= c + 1;`
	`end`

wait statements	`wait(clk);`

Named events	`event my_event;`
	`initial   -> my_event;`
	`always begin`
	`  @(my_event);`
	`  a <= b;`
	`end`

Static SystemVerilog data types	`logic, bit, int, longint, byte, void, struct, enum`
Block disable within a process	-
Delta delays for races	`zemi3_event advanced property`

Table 1 Supported Sub-Set of Behavioral Code (Continued)

Supported Constructs	**Example**
System tasks	• `$display` • `$fdisplay` • `$writeh` • `$fclose` For a complete list of supported system tasks, see List of Supported System Calls.

Clocks, Multiple Clocks, and Multiple Edges

Any combination of clocks and edge expressions are supported in transactors, even inside export tasks. It is possible to mix, in the same process, multiple clocks and multiple edges of the same clock. For example, inserting statements like `@(posedge clock)` or `@(negedge clock)` in the blocks. It is also possible to use such statements to describe a sequence of actions that execute as an implicit state machine.

Note:

Do not mix controlled clocks and sampled clocks in the same event expression.

Usage of Blocking and Non-Blocking Assignments

Mixing blocking and non-blocking assignments to the same register is not allowed and results in a compiler error.

Non-blocking assignments should not be used for function outputs or return values since functions cannot consume time. However, they can be used for side effects.

Limitations

The following features are not supported in ZEMI-3:

- SystemVerilog data types other than `logic, bit, int, longint, byte, void, struct, enum`.
- `fork/join`
- `cmos, nmos, pmos, rcmos, rnmos, rpmos`
- Strengths on drivers
- Delays for behavior

- System tasks/functions other than:
  - `$display`, together with: `$displayh, $displayb, $displayo`
  - `$fdisplay`, together with: `$fdisplayh, $fdisplayb, $fdisplayo`
  - `$write`, together with: `$writeh, $writeb, $writeo`
  - `$fwrite`, together with: `$fwriteh, $fwriteb, $fwriteo`
  - *$fopen; $fclose*
- Procedural assign/deassign
- Inter-process block disable and task disable

Adding Delta Delays

It is possible to add to the hardware part of the transactor delta delays clocked by the ZeBu `driverClock`, the internal high-speed clock. Because delta delays introduce sequential elements, they can be used to tradeoff performance for hardware size. Since the frequency of the `driverClock` is higher than any design clock in the system, delta delays are shorter than the shortest design clock cycle.

Adding delta delays in loops can reduce the hardware size, but they reduce the speed.

A delta delay is specified using the zemi3_event pragma in an empty statement:

```
(*zemi3_event*);
```

Be sure to add the semi-colon; otherwise, you may experience incorrect runtime behavior.

You can also use delta delays to split long combinational paths, or to reduce the number of ports required for a memory, as in the following example:

```
reg [31:0] mem [0:1023]
integer i;
for (i=0;i<1024;i=i+1) begin
(*zemi3_event*);
mem[i] = 0;
end
```

In this case, the 1024-port memory requires a multiport if synthesized without delta delays. The insertion of delta delays at each access transforms the memory into a single-port memory with 1024 consecutive accesses, but the performance is lower since it requires 1024 `driverClock` cycles to execute the loop.

Supported System Calls

This section has the following two subsections:

- List of Supported System Calls
- Enabling System Calls

List of Supported System Calls

The following system calls are supported in the source code of a ZEMI-3 transactor:

- `$display`, together with: `$displayh, $displayb, $displayo`
- `$fdisplay`, together with: `$fdisplayh, $fdisplayb, $fdisplayo`
- `$write`, together with: `$writeh, $writeb, $writeo`
- `$fwrite`, together with: `$fwriteh, $fwriteb, $fwriteo`
- `$fopen`
- `$fclose`

All these system calls are serialized with respect to each other (so they execute in order and share resources), but they remain independent (not serialized) with respect to DPI calls.

Enabling System Calls

To enable system calls, you must add the following line to the ZEMI-3 attribute file before compiling:

```
zemi3 -support_dollar_display true
```

Advanced Properties

The following table lists the SystemVerilog pragmas that denote advanced properties that can be declared in the hardware part of a transactor to modify or optimize its behavior:

Table 2 Advanced Properties

Advanced Properties	Example
(zemi3_event *) ;*	Inserts one delta delay (see Usage of Blocking and Non-Blocking Assignments).

Table 2 Advanced Properties (Continued)

Advanced Properties	**Example**
(zemi3_stream=0 *)*	Disables streaming mode on import/export (see Streaming Data and Forcing/Disabling Streaming Mode).
(zemi3_highpriority *)*	Sets the ports associated with import/export to high priority.
(zemi3_protocol=<my_protocol> *)*	Specifies the name of the protocol when exported tasks share critical resources (see Protocol Mode).
(zemi3_serialize=<my_serializer> *)*	Specifies the name of the serializer when imported tasks are serialized (see Serializing Calls and Serialization).
(zemi3_call_skipped = "fail" *)*	Specifies the import to be a lossy call and the name of the bit indicating if the import is performed or not (see Lossy Calls).
(zemi3_bufferedport[=<size>] *)*	Specifies the import to be buffered and gives the size of the buffer (see Port Buffering).

This section has the following subsections:

- Forcing/Disabling Streaming Mode
- Protocol Mode
- Serialization
- Lossy Calls
- Port Buffering

Forcing/Disabling Streaming Mode

If you do not want ZEMI-3 to optimize DPI calls into the streaming communication, you can disable streaming mode by specifying `zemi3_stream=0` in the DPI declaration:

```
(* zemi3_stream=0 *)
import "DPI-C" function int start_test ();
```

By default, context import functions are not optimized for streaming because they may call export functions. If the context import implementation contains no export call, the streaming mode can be enabled by specifying `zemi3_stream=1` in the DPI declaration:

```
(* zemi3_stream=1 *)
import "DPI-C" context function int start_test ();
```

Protocol Mode

To declare export tasks for a given protocol, you must specify the same advanced property (`zemi3_protocol=<protocol_name>`) before the declaration of each export task that belongs to the protocol.

For example:

```
(* zemi3_protocol="dut_inc_dec", zemi3_stream=0 *)
export "DPI-C" task inc_dut_input;

(* zemi3_protocol="dut_inc_dec" *)
export "DPI-C" task dec_dut_input;
```

Serialization

To serialize import calls, use the `zemi3_serialize=<serializer name>` advanced property before the declaration of each import task that belongs to the same serializer.

For example:

```
(* zemi3_serialize="mem_read_write" *)
import "DPI-C" context function int mem_read_addr (int addr);

(* zemi3_serialize="mem_read_write" *)
import "DPI-C" context function void mem_write_addr (int addr, int data);
```

Lossy Calls

The lossy calls feature allows you to skip an import call when the communication channel between the hardware and the software is busy. Without the lossy calls feature, the import would block the hardware part until the communication channel becomes available.

To determine if the call has executed, an artificial output bit must be added at the end of the argument list. This bit is set to 1 after the call to indicate that the call did not execute.

To specify the import to be a lossy call, the following property has to be used:

```
(* zemi3_call_skipped = "fail" *)
import "DPI-C" function void send_result(input bit [127:0] dout, output
 bit fail) ;
```

For example:

```
module xtor(input CLK,
            input [31:0] dout) ;
  integer i ;

  reg [31:0] failures ;
  bit fail ;


  (* zemi3_call_skipped = "fail" *)
  import "DPI-C" function void send_result(input bit [127:0] dout,output
 bit fail) ;

  initial failures = 0 ;

  always @(dout) begin
    send_result(dout, fail) ;
    if (fail == 1) failure = failure + 1 ;
  end

  /* ... */

endmodule
```

Port Buffering

For streaming import calls, it is not necessary for the software to see the effect of the import call immediately. Instead, the cost of sending many small messages can be saved by accumulating a number of these messages and sending them as a single block.

Sending fewer messages allows you to increase the global frequency of the controlled clocks. However, this increases the latency between the hardware call and the import function call.

This can be done only for imports that are streaming and not part of any serializer (named or anonymous).

```
(* zemi3_bufferedport[=<size>] *)
```

where <*size*> is the size of the final accumulated block. This value is optional - if it is not given, a default value is used (which can be modified in the UTF file). For example:

```
//Create a buffer of 31 import calls (buffer is 4096 for 128-bit
 messages)
  (* zemi3_bufferedport=4096 *)
  import "DPI-C" function void send_result(input bit [127:0] dout)
```

6

Writing the Software Part

Calling Export Functions/Tasks in C

Export functions (and tasks) are SystemVerilog functions (and tasks) that are called from the software part of the transactor (see Import Function).

The prototype of the export function is available in the ZEMI-3 transactor header file (`<my_xtor>.h`) generated by **zCui** during compilation of the hardware part of the ZEMI-3 transactor (see Compiling Output Files for the Software Part of the Transactor). In the software part of the transactor, the export function is called as any other C function:

```
for(uint i=0;i<10;i++) {
  write_one_addr(i, data[i]);
}
```

Before calling an export function, the current scope has to be properly set to the instance path of the export function in the transactor. This is done using the standard SystemVerilog DPI utility function `svSetScope`.

For example:

```
 /* Setting the scope of the export call */
  svScope s = svGetScopeFromName("top.xtor0");
  svSetScope(s);
```

Prefetch Mode

In prefetch mode (see Prefetch Calls), the import function is not called with its SystemVerilog name. The following specific functions are generated for each import function having only outputs:

- The `block_prefetch_<import_name>` function provides a blocking mode.
- The `try_prefetch_<import_name>` function provides a non-blocking mode.

Enabling or Disabling the Prefetch Mode

Enable the prefetch mode (the original import function is not called while prefetching is enabled) be declaring the function as follows:

```
extern "C" void enable_prefetch_<import_name> ()
```

Disable the prefetch mode as follows:

```
extern "C" void disable_prefetch_<import_name> ()
```

Blocking Prefetch Functions

Calling the `block_prefetch_*` function blocks the prefetch function when the message buffer is full. It is recommended to call the `block_prefetch_*` function in a separate thread, so that the main processing is never blocked. This function always sends the message (once the message buffer becomes available) and returns true.

```
bool block_prefetch_<import_name> ([<return_value>,] <import args>)
```

where,

- `<import_name>` is the original name of the import function.
- `<import args>` is the list of arguments of the import function.
- `<return_value>` is the value returned by the import function if any.

Non-Blocking Prefetch Functions

The `try_prefetch_*` and `block_prefetch_*` functions are similar, except that `try_prefetch_*` does not send the message when the message buffer is full, and returns false. It returns true if the message is sent.

```
bool try_prefetch_<import_name> ([<return_value>,] <import args>)
```

where,

- `<import_name>` is the original name of the import function.
- `<import args>` is the list of arguments of the import function.
- If the import function returns a value using the return mechanism, `<return_value>` is this value returned by the import function.

zemi3_ExportCallIsBlocked() Macro

This macro can be used to determine if an export function executes without blocking the software part. For example, if the hardware part is not ready to receive any message at this time, the macro returns true once the hardware part is ready to execute the export call. For example,

```
//...
while (zemi3_ExportCallIsBlocked(xtor0_send)) {
xtor1_get_data(data);
screen_display(data);
}
// Now the export is ready to be executed
xtor0_send(command);
//...
```

Implementing Import Functions in C

An import function (or task) is a C function that is called from the hardware part of the transactor (see Import Function). The import function must be implemented as an extern "C" function:

```
extern "C" void read_addr(const uint *addr, const uint *data)
{
  *data = array[*addr];
}
```

Note:

All import functions are resolved in the global scope.

In a ZEMI-3 transactor, calls from anywhere in the hardware scopes call the same import function (same name import). If the code needs to understand the call context, then the import should be declared as context. In this case, the SystemVerilog function `svGetScope()` can be used to determine the context of the call.

C Types Supported by ZEMI-3

The DPI standard defines the mapping between SystemVerilog types and their C counterparts. The ZEMI-3 transactor does not use all of them, but supports the main ones.

Table 3 Mapping Between SystemVerilog and C Language Types

DPI Formal Argument Types	Corresponding Types Mapped to C
`byte`	`char`

Table 3 Mapping Between SystemVerilog and C Language Types (Continued)

DPI Formal Argument Types	Corresponding Types Mapped to C
`byte unsigned`	`unsigned char`
`shortint`	`short int`
`shortint unsigned`	`unsigned short int`
`int`	`int`
`int unsigned`	`unsigned int`
`longint`	`long long`
`longint unsigned`	`unsigned long long`
Scalar values of bit type	`unsigned char`
Packed one-dimensional arrays of both:bit typeslogic typesPacked multidimensional arrays are also supported, but they are presented as single dimensional arrays on the C-side.	Canonical arrays of both:`svBitVecValsvLogicVecVal`

The specific SystemVerilog types are declared in the standard header file, which can be found in `$ZEBU_ROOT/include/svdpi.h`.

SystemVerilog C Functions Supported by ZEMI-3

The functions mentioned below are declared in the standard header file, which can be found in `$ZEBU_ROOT/include/svdpi.h`.

The following table lists the SystemVerilog standard DPI utility functions supported in the ZEMI-3 environment (for more details, see the SystemVerilog LRM):

Table 4 Supported SystemVerilog DPI Utility Functions

Prototype	Description
`svScope svGetScope(void)`	Gets the scope of called imported function
`svScope svSetScope (const svScope scope)`	Sets the scope for exported function

Table 4 Supported SystemVerilog DPI Utility Functions (Continued)

Prototype	Description
`int svPutUserData (const svScope scope, void *userKey, void *userData)`	Associates a data and a scope
`void *svGetUserData (const svScope scope, void *userKey)`	Gets data associated to a scope
`const char *svGetNameFromScope (const svScope scope)`	Gets the scope name
`svScope svGetScopeFromName (const char *scopeName)`	Gets the scope from its name
`int svGetCallerInfo (char **fileName, int *lineNumber)`	Gets the line number and file name of the caller
`const char *svDpiVersion(void)`	Gets the DPI version number
`svBit svGetBitselBit (const svBitVecVal *s, int i)`	Gets a bit from a `svBitVecVal`
`void svPutBitselBit (svBitVecVal *d, int i, svBit s)`	Puts a bit in a `svBitVecVal`
`void svGetPartselBit (svBitVecVal *d, const svBitVecVal *s, int i, int w)`	Gets a part of a `svBitVecVal`
`void svPutPartselBit (svBitVecVal *d, const svBitVecVal s, int i, int w)`	Puts a part of `svBitVecVal` in another `svBitVecVal`

Controlling the Back-to-Back Mode From Your Testbench

With the back-to-back mode you can stop the clocks of a specified transactor between calls to export functions. This mode can be enabled at compilation or runtime and can be disabled at runtime.

The following table lists C functions available for back-to-back control.

Table 5 Back-to-Back Mode Functions

Function	Description
`void ZEMI3_startBack2Back const char *xtorName)`	Starts the back-to-back mode for the specified transactor.
`void ZEMI3_stopBack2Back const char *xtorName)`	Stops the back-to-back mode for the specified transactor.
`bool ZEMI3_isBack2BackStarted (const char *xtorName)`	Returns true if the back-to-back mode is started for the specified transactor.

Feedback

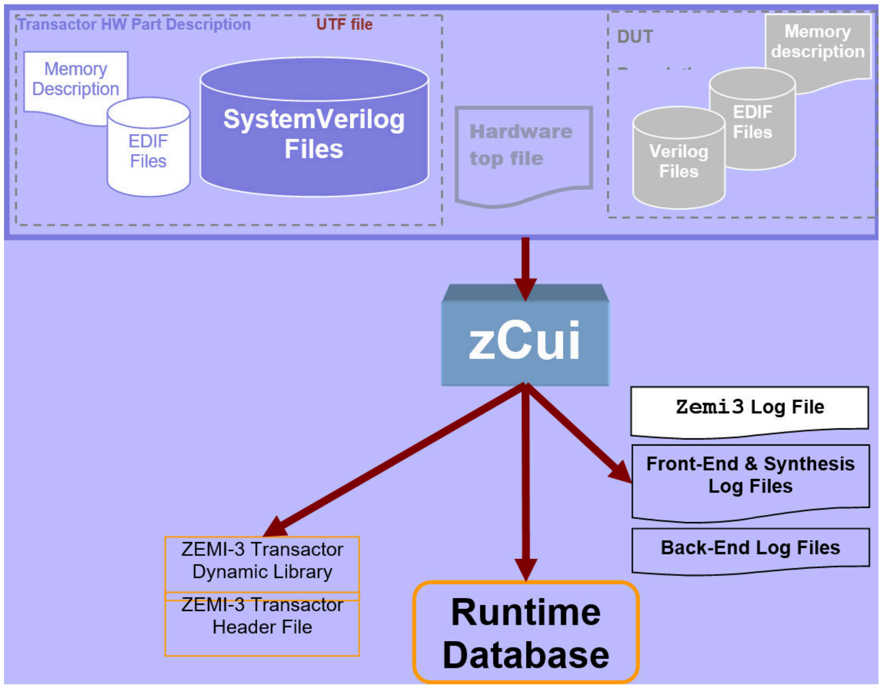
7

Compiling the Hardware Part of a ZEMI-3 Transactor

The hardware part of a ZEMI-3 transactor is compiled from its SystemVerilog description, and, optionally, EDIF netlists and zMem memory descriptions.

The hardware part of a ZEMI-3 transactor is compiled by **zCui**'s Unified Compile as illustrated in the following figure.

Figure 9 Compiling the Hardware Part of a ZEMI-3 Transactor



The transactor is first processed by VCS to identify all the export and import functions and tasks. This operation is done before launching the design front-end for synthesis.

This compilation process requires a hardware top in which the transactor is connected to the DUT and the controlled clocks are declared.

The compilation of the software part of the transactor is different according to the runtime environment.

Instantiating a ZEMI-3 Transactor in the Hardware Top

A ZEMI-3 transactor can be instantiated anywhere in the design hierarchy beneath the hardware top-level (`hw_top`). The transactor is instantiated with its connections to the DUT in a SystemVerilog compliant way.

In a ZEMI-3 transactor, the clock is declared like any other port of the transactor. The clock must be a controlled clock generated by ZeBu, accordingly, the clock port of the transactor is connected to the output of a `zceiClockPort`.

```
For example:
streamer_out s0(
  .clk(clk),
  .data(count)
);

zceiClockPort c(
  .cclock(clk)
);
```

Note:

A transactor top module cannot have the name “reset”. This is a known limitation and it is recommended to use a name other than “reset”.

Adding and Setting a ZEMI-3 Transactor in the UTF File

This section consists of the following subsections:

- Adding a ZEMI-3 Transactor
- Setting the Transactor

Adding a ZEMI-3 Transactor

To add a ZEMI-3 transactor in the compilation project, use the following UTF command:

```
xtors -add {list of xtor_module_names} -type ZEMI3
```

For a detailed description, type “*vcs –help utf+xtors*”.

The following table lists ZEMI-3 options available in the UTF file. For command line help, type “`vcs -help utf+zemi3`”.

Table 6 ZEMI-3 Options available in the UTF File

Function	Description
`*-module {list_of_zemi3_transactors_modules}`	Provides a list of ZEMI-3 transactor modules.
`*-instance <xmr path to xtor> -disable <xmr to signal> ]`	Specifies a disable signal for a ZEMI-3 transactor instance.
`*-allow_streaming <bool>`	Disables streaming for the transactor when this is set to false.
`*-start_in_back_to_back <bool>`	Starts run in back to back mode (clocks do not progress until an export is called.).
`*-support_dollar_display <bool>`	Adds support for $display.
`*-do_profiling <bool>`	Enables profiling.
`*-hard_max_out_port_width <int>`	Sets the maximum ZCEI output port width.
`*-hard_max_in_port_width <int>`	Sets the maximum ZCEI input port width.
`*-auto_flush_interval <int>`	Sets the auto-flush period for buffering in *driverclock* cycles.

`*-port_optimization_mode 0|1|2|3`
`0:AREA_OPT`
`1:THROUGHPUT_OPT`
`2:LATENCY_OPT`
`3:DEFAULT_OPT`
	Sets the port optimization mode.

`*-profile_counters_width <int>`	Sets the width of profiling counters.
`*-all_tf_args_are_automatic <bool>`	Gives control over task/function arguments. By default, it is set to false.When task/function is duplicated (inlined): If `all_tf_args_are_automatic` is false, the static variables need to be synchronized across copies. Otherwise, they are considered automatic and no synchronization is needed.
`*-external_controlled_clocks {list_of_external_controlled_clocks}`	Specifies a list of additional control clocks for the transactor.
`* -allow_mixed_design_clocks <bool>`	Allows a mix of sampled clocks and controlled clocks in the same blocks.

Table 6 ZEMI-3 Options available in the UTF File (Continued)

Function	Description
`*-exports_excluded_from_back_to_back {list_of_export_DPI_excluded_from_back_to_back_mode}`	Lists exports excluded from back to back.
`*-max_loop_iterations <int>`	Controls the maximum size of combinational iterating loops in ZEMI-3 synthesis.
`*-max_clocked_loop_iterations <int>`	Controls the maximum size of sequential iterating loops in ZEMI-3 synthesis.
`*-merge_multiple_comb_writers <bool>`	Handles multiple drivers of the same variable from different blocks. Writes are serialized.
`*-default_cclk {xmr to signal that is connected to zceiClockport}`	Sets the default controlled clock if the compiler is unable to select it automatically.
`* -sample_clock_mode <bool>` **Note:** `-sample_clock_mode` is a global option. Other `-module|-instance` options cannot be used with `-sample_clock_mode`.	Enables sampled clock optimization mode to avoid sampling lock effect.
`* -timestamp <bool>` **Note:** `-timestamp` can be used with or without `-module`.	Allows using `svGetTimeFromScope` in DPI imports.

Setting the Transactor

Setting for the Streaming Option (Optional)

Streaming mode improves performance by optimizing single-direction data exchanges (see also *Streaming Data* in Data Exchange Between Hardware and Software and *Forcing/Disabling Streaming Mode* in Writing the Hardware Part). It is enabled by default; however, you can disable it using the following UTF command:

```
zemi3 -module {list_of_zemi3_transactors_modules} -allow_streaming false
```

Setting for the Back to Back Mode (Optional)

Running your emulation in back to back mode eases the debug analysis of ZEMI-3 transactors. For more details, see Back-to-Back Mode.

To enable the back to back mode, add the following command in the UTF file:

```
zemi3 -module {list_of_zemi3_transactors_modules} -start_in_back_to_back
 true
```

Defining Advanced Settings (Optional)

Some advanced settings for ZEMI-3 transactors can be set in the UTF file. The following option can be used for automatic optimization:

```
zemi3 -module {list_of_zemi3_transactors_modules}
      -port_optimization_mode 0|1|2|3

0 for AREA_OPT  2 for LATENCY_OPT
1 for THROUGHPUT_OPT 3 for DEFAULT_OPT
```

where,

- `AREA`: Minimizes FPGA resources by factoring the usage of hardware ports for imports and exports. This option also reduces, whenever feasible, the message size by splitting the message in sequential packets (multicycle).
- `LATENCY`: Minimizes the exchange time between hardware and software.
- `THROUGHPUT`: Maximizes data throughput by buffering streaming import messages. You can use the `zemi3_highpriority` property (see Supported System Calls) when a transactor contains an import that needs maximum throughput. Also, user-defined optimizations, specific to the message multiplexing and message buffering, can be specified in the UTF file with zemi3 option.
- Use Message Multiplexing: Activates message multiplexing for software/hardware and hardware/software communication with the maximum port width in the associated field. The default values are 8096 for software/hardware, and 8160 for hardware/software.
- Use Message Buffering: Activates streaming for imports, with the maximum message port width in the associated field (the default value is 8096). The port width declared here is the default value used for message port buffering, which can be modified with `zemi3_bufferedport` property (see *Advanced Properties* in Writing the Hardware Part).

Compiling a ZEMI-3 Transactor with zCui

If you have all the sources of a ZEMI-3 transactor and DUT, you can compile with zCui using the following command.

```
zCui -u <project.utf> <other options>
```

Results of the ZEMI-3 Transactor Pre-Processing

The log information can be found at the following location:

```
zcui.work/xtor_<my_xtor>/zxtor/<my_xtor>/zxtor.log
```

The logs provide information about:

- Each inter-language functions. For each import/export function, the elements are listed in the following table:

Table 7 Log File Elements

Log File Element	Description
Name	Name of the function/task
Type	Function or task
in bits	Total size of input bits
out bits	Total size of output bits
Protocol	Name of the protocol belonging to the task (if defined)
streaming	Activation of the streaming (yes|no)
Context	Activation of the context (yes|no)
prefetchable	Activation of prefetching (yes|no)

- Message ports intended for advanced optimization of transactional environment.

  For example:

```
INFO: DPI exports summary:
INFO: +-----------+------+---------+----------+----------+--------+
INFO: |      name | type | in bits | out bits | protocol | streaming |
INFO: +-----------+------+---------+----------+----------+--------+
INFO: | stream_in | task |      32 |        0 |       -- |       Yes |
INFO: +-----------+------+---------+----------+----------+--------+
INFO:
INFO: DPI imports summary:
INFO: +------+----------+---------+----------+---------+---------+
INFO: | name |     type | in bits | out bits | context | streaming |
 prefetchable |
INFO: +------+----------+---------+----------+---------+---------+
INFO: +------+----------+---------+----------+---------+---------+

INFO:
INFO: Message Ports Summary:
INFO: +-------------------------------+------+------+-------+----+
INFO: |                          name | type | size | count |
 multiplex |
INFO: +-------------------------------+------+------+-------+----+
INFO: | streamer_in.stream_in_in_port |   in |   32 |     1 |       |
INFO: |       streamer_in.b2b_in_port |   in |    1 |     1 |        |
INFO: |      streamer_in.b2b_out_port |  out |    1 |     1 |        |
INFO: +-------------------------------+------+------+-------+----+
INFO:
INFO: Total Message In Ports: 2 (33 bits)
INFO: Total Message Out Ports: 1 (1 bits)
```

Compiling Output Files for the Software Part of the Transactor

ZeBu compilation generates two types of output files, described in the following sections. This section consists the following subsections:

- Dynamic Libraries
- Compiling Transactor Libraries
- Header File

Dynamic Libraries

For each ZEMI-3 transactor, the dynamic library must be created before starting runtime using the following commands.

```
make -f dpixtor.mk -C $(ZCUIWORK)/$(ZEBUWORK) clean all
    or
cd <zcui.work>/zebu.work
make -f dpixtor.mk clean all
```

Note:

These commands must be run on a PC running the same OS as the runtime host.

These commands must be re-executed if the OS of the runtime host changes.

Compiling Transactor Libraries

To compile the transactor libraries, add the following additional option:

```
make -C $(ZEBUWORK)/backend_default -f dpixtor.mk CUSTOM_DIR=1
```

If the `CUSTOM_DIR=1` option has passed to compile, transactors libraries are generated at `<current_dir>/zebu_zemi3_libs`.

`zebu_zemi3_libs` helps ZEMI-3 runtime to locate these libraries automatically.

At runtime, the transactor libraries are searched at `zebu_zemi3_libs` first. If this directory/ transactor library is not found, the search continues to `zcui.work/zebu.work` and `zcui.work/zemi3`, respectively.

Header File

For each ZEMI-3 transactor, the ZeBu compiler generates a <my_xtor>.h header file where <my_xtor>is the name of the ZEMI-3 transactor instantiated in the hardware top.

The header file is stored in the `zcui.work/zebu.work` directory.

This header file declares the following:

- The prototypes of the import and export functions of the transactor.
- The user transactor interface class, derived from the base class ZEMI-3 transactor.

This header file must be included in the software part of the transactor (see Writing the Software Part) and in the testbench (see Testbench Function).

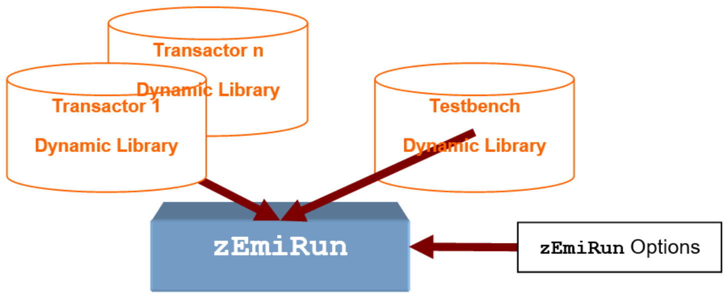
8

Running Emulation of ZEMI-3 Transactors

The ZEMI-3 environment is very similar to a simulation environment. A generic executable called **zEmiRun** handles the emulation automatically. You must provide the implementation of the import calls and optionally a testbench to control the emulation.

A testbench is useful for export tasks that consume simulation time. The following figure illustrates the ZEMI-3 Runtime Environment.

Figure 10 ZEMI-3 Runtime Environment



The following advanced runtime environments can also be used with a ZEMI-3 transactor

- An existing SystemC software testbench environment (see Using a SystemC Testbench).
- A ZCEI transactors along with ZEMI-3 transactors (see Using ZCEI Transactors and ZEMI-3 Transactors).

Prerequisites

The following are the prerequisites to run the ZEMI-3 environment:

- **zEmiRun** is a multithreaded application, thus, it is highly recommended to run it on a multiprocessor host to get the best performance.
- If the compilation is performed on a host configured differently from the one used for emulation runtime, the dynamic libraries can be independently recompiled with the

`dpixtor.mk` `makefile` in the `zebu.work` directory. This `makefile` has two different targets:

  - *all*: Compiles all dynamic libraries and copies them to `zebu.work`.
  - *clean*: Cleans all the dynamic libraries.

- The following options can be used to override the makefiles generated by the compilation, if necessary:
  - *LDFLAGS* to add linker options [default: *empty*].
  - *CXXFLAGS* to add compiler options [default: -*O*].
  - *ZEMI3_LIB* to change the ZEMI-3 library [default: *ZebuZEMI3*].
  - *XTOR_LIB* to rename the generated `*.so` file [default: `<xtor_mod_name>.so`].
- Ensure that none of the dynamic libraries loaded by **zEmiRun** are explicitly linked to `libZebu.so` or to `libZebuThreadsafe.so`. You can check using `ldd <libname>.so`.
- Ensure to remove all `-lZebu` and `-lZebuThreadsafe` when creating the `.so` file.

Preparing the Files Required for Runtime

You must provide the following for ZEMI-3 runtime:

- Implementation of the import calls.
- Implementation of the testbench function.

The import and export functions and the testbench function must be declared as extern "C".

The source of the testbench must include the standard header file for SystemVerilog DPI (`svdpi.h`) and the transactor header files generated by the compilation (`<my_xtor>.h`) (see Compiling Output Files for the Software Part of the Transactor).

This section consists of the following subsections:

- Testbench Function
- Compilation of the Dynamic Library

Testbench Function

You can provide a C function that executes export calls to control the emulation. This testbench function is called by the `ZEMI3Manager` object. When this function returns, the emulation terminates.

This testbench function must be declared as extern “C”. It can call export tasks that consume simulation time.

The testbench function prototype is as follows:

```
int my_testbench_function (int argc, char **argv)
```

where, the `int` is the returned status of `zEmiRun`.

If no testbench function is provided and barring some other termination mechanism (for example, fixed time or number of clock cycles), the emulation runs indefinitely.

Compilation of the Dynamic Library

Your dynamic library must be linked with the `libZebuZEMI3.so` library.

The `-rdynamic` and `-fPIC` options should be used when linking with C++ code that contains import functions.

The following example displays how to compile dynamic library:

```
$ g++ -fPIC -c user_testbench.cc -Izcui.work/zebu.work/
  -I$ZEBU_ROOT/include
$ g++ -fPIC -rdynamic -shared -o user_testbench.so user_testbench.o \
  -L$ZEBU_ROOT/lib -lZebuZEMI3
```

ZEMI-3 Support With zRci

zRci supports ZEMI-3 environment. For details, see the *ZeBu Unified Command-Line User Guide*.

9

Running Emulation in a Heterogeneous Environment

The following advanced runtime environments can be used with a ZEMI-3 transactor:

- An existing SystemC testbench (see Using a SystemC Testbench).This mode gives access to a *ZEMI3Manager* object for emulation control.
- A zcei transactors along with ZEMI-3 transactors (see Using ZCEI Transactors and ZEMI-3 Transactors).

  Note:

  If you need to run your ZEMI-3 transactors in a 32-bit environment, contact your local representative.

Using a SystemC Testbench

In a SystemC environment, the runtime is handled by the SystemC kernel. Since **zEmiRun** uses its own scheduler similar to SystemC, they cannot be used together. In a SystemC environment, you must explicitly initialize the ZEMI-3 environment.

In such an environment, your testbench should use a C++ class that initializes the transactors, as described in the following sections.

This section has the following subsections:

- Initializing the ZEMI-3 Environment Using the ZEMI3Manager Class
- Initializing the ZEMI3Manager Object

Initializing the ZEMI-3 Environment Using the ZEMI3Manager Class

The `ZEMI3Manager` class initializes the ZEMI-3 environment. The interface of this class can be found in the $ZEBU_ROOT/include/ZEMI3Manager.hh file.

The following sections describe the procedure for environment initialization.

Creating the ZEMI3Manager Object

Create the *ZEMI3Manager* object by calling the open method as follows:

```
ZEMI3Manager* ZEMI3Manager::open (const char *zebu_work,
                                  const char *designFeatures,
                                  const char *processName");
```

where,

- *zebu_work*: Path to the working directory.
- *designFeatures*: Path to the designFeatures file.
- *processName*: The name of the process.

  Note:

  You can open only one ZEMI3Manager object.

For example:

```
ZEMI3Manager *dm = open ("./zebu.work", ". /designFeatures",
 "myProcess");
```

Declaring the List of ZEMI-3 Transactors

A list of transactors is automatically generated in the zebu.work directory as `xtor_dpi.lst`. You must specify this list to the runtime environment as follows:

- When using a single-process, that is, only one ZEMI-3 transactor process is defined in your `designFeatures` file. Add the following line to your testbench:

  ```
  void buildXtorList(const char *xtor_dpi.lst);
  ```

-

By default, this command uses the automatically generated `xtor_dpi.lst`.

For example:

```
manager->buildXtorList(xtor_dpi.lst);
```

or

```
manager->buildXtorList();
```

- When using multiple processes, that is, multiple ZEMI-3 transactor processes are defined in your `designFeatures` file (therefore as many testbenches as the number of processes). You must do the following:

Manually split this `xtor_dpi.lst` list to create one `.lst` file per process.

Add the following line to each of your testbenches that declare separate lists for the runtime environment:

```
void buildXtorList(const char *<xtor_list_custom_n.lst>);
```

where, `<xtor_list_custom_n.lst>` is the name of the customized transactor list.

For example:

```
manager->buildXtorList(my_xtor_list01);
```

Initializing the ZEMI3Manager Object

The initialization starts the clocks and launches a thread for each transactor. The launched threads call the transactor imports. Start the initialization as follows:

```
dm->init();    // dm is the ZEMI3Manager object
```

Getting the ZEBU::Board Object With a ZEMI3Manager Object

When runtime features, such as, access to signals, logic analyzer, or clock control are required, you can retrieve the *ZEBU::Board* object from the *ZEMI3Manager* object as follows:

```
ZEBU::Board* getBoard();
```

For example:

```
Board *z = dm->getBoard();
```

Controlling the Back-to-Back Mode

It is possible to control the back-to-back mode from the ZEMI3Manager object.

To do so, use the functions described in Controlling the Back-to-Back Mode From Your Testbench.

Handling Errors Inside the ZEMI3Manager Object

Capture Exceptions

When errors are detected inside the `ZEMI3Manager` object, an exception is generated. This exception must be captured with a try-catch statement.

For example:

```
try{
  // initialization code here
}
catch (exception &xcp) {
  printf("###  aborting %s - fatal error : %s.", argv[0], xcp.what());
  exit(1);
}
catch (...) {
  printf("###  aborting %s - fatal error... ", argv[0]);
  exit(1);
}
```

Compile Libraries for the ZEMI-3 Environment

The final executable must be linked with the following libraries:

- The ZEMI-3 library: libZebuZEMI3.so with the `-lZebuZEMI3` option.
- The ZeBu library: libZebu.so or libZebuThreadsafe.so with the `lZebu` or `-lZebuThreadsafe` option respectively.

The libZebuThreadsafe.so library must be used with a multithreaded testbench.

It is recommended to use the libZebuThreadsafe.so library first if you want to perform your design bring-up safely. After you get the expected behavior, you can perform the bring-up again with the libZebu.so library to improve performance.

For example:

```
$ g++ -fPIC -c dpi_ctrl.cc -Izcui.work/zebu.work -I$ZEBU_ROOT/include
$ g++ -rdynamic -o user_tb *.o -L$ZEBU_ROOT/lib \
 zcui.work/zebu.work/my_xtor_1.so zcui.work/zebu.work/my_xtor_2.so \
 -lZebuZEMI3 -lZebuThreadsafe
```

Using ZCEI Transactors and ZEMI-3 Transactors

It is possible to mix ZEMI-3 transactors and ZCEI transactors. In this case, you have a ZCEI-based environment or a ZEMI-3 based environment. As in standard ZCEI-based environments, all ZeBu initializations are done through the `ZEBU::Board` class.

The programming interface for ZEMI-3 transactors is defined by a base class in the ZEMI3Xtor.hh header file located in the $ZEBU_ROOT/include directory.

The ZEMI-3 transactor base class lets you do the following:

- Connecting and disconnecting a ZEMI-3 transactor.
- Serving the imports.
- Starting and stopping back-to-back mode.

This section has the following subsections:

- Using the ZEMI3Xtor Class
- Using a ZEMI-3 Transactor in a ZCEI Testbench
- Running a ZEMI-3 Transactor

Using the ZEMI3Xtor Class

A testbench for the ZEMI-3 transactor must start with the following lines:

```
#include "ZEMI3Xtor.hh"
using namespace ZEBU
```

The ZEMI3Xtor.hh file defines the `ZEMI3Xtor` class.

The `ZEMI3Xtor` class represents the ZEMI-3 transactor object. The methods associated with this class are described in the following table.

Table 8 ZEMI3Xtor Class Methods

Method	Description
`ZEMI3Xtor`	Constructor.
`~ZEMI3Xtor`	Destructor.
`init`	Initializes the ZEMI-3 transactor.
`close`	Closes the ZEMI-3 transactor.
`needsService`	Checks if any ZeBu imports need service.
`reset`	Restarts the processing of imports/exports.
`terminate`	Terminates any ongoing processing, including service loop.
`setGroup`	Specifies a group ID number for the ZEMI-3 transactor.
`registerImports`	Registers imports in *Board::serviceLoop*.
`unregisterImports`	Unregisters imports in *Board::serviceLoop*.
`enableImports`	Enables calling the streaming imports.
`disableImports`	Stops calling the streaming imports.
`isImportDisabled`	Indicates whether calling the streaming imports is enabled or not.

Table 8 ZEMI3Xtor Class Methods (Continued)

Method	Description
`checkImports`	Checks whether there is any message pending for any of the imports.
`serviceLoop`	Calls the ZEMI-3 transactor service loop.
`serviceLoopWithWait`	Calls the ZEMI-3 transactor service loop and returns only when an import call occurs or after the specified timeout.
`startBackToBack`	Starts the back-to-back mode for the ZEMI-3 transactor.
`stopBackToBack`	Stops the back-to-back mode for the ZEMI-3 transactor.
`isBackToBackStarted`	Indicates whether the back-to-back mode is started or not.
`flushBufferedPorts`	Flushes messages of the ports that are automatically buffered.
`getXtorInfos`	Returns information about the import from which this method is called.
`getErrorStatus`	Indicates whether an error is detected inside the ZEMI-3 transactor.
`getExportActiveStatus`	Indicates whether an export task from the ZEMI-3 transactor is currently running.

init() Method

This method initializes the ZEMI-3 transactor and activates its ports. It should be called before the ZeBu `Board::init` initialization.

```
void init (Board *zebu, const char *instancePath) ;
```

where,

- zebu is the pointer to the `ZEBU::Board` object.
- `instancePath` is the path to the instance of the ZEMI-3 transactor.

close() method

This method closes the ZEMI-3 transactor and deactivates its ports.

```
void close () ;
```

needsService() Method

This method checks whether there are any ZeBu imports that need service through a service loop.

```
bool needsService () const;
```

This method returns the following:

- *true*: Import needs service.
- *false*: No import needs for service.

reset() Method

This method re-starts the processing of imports/exports after a terminate method.

```
void reset () ;
```

terminate() Method

This method terminates any ongoing processing. This also stops any ongoing service loop.

```
void terminate () ;
```

setGroup() Method

This method sets a group ID number for the ZEMI-3 transactor. The group ID can be used with `Board::serviceLoop` to serve the imports.

```
void setGroup(unsigned int portGroup);
```

where, `portGroup` is the group ID number.

registerImports() Method

This method allows registering imports in `Board::serviceloop`. Thus, when `Board::serviceloop` is called, the ZEMI-3 transactor's imports are called.

```
void registerImports(unsigned int portGroup);
```

where, `portGroup` is optional. It is the group ID number used in `Board::serviceloop` if any. It allows associating a specific import to the specified group of transactors.

unregisterImports() Method

This method unregisters the imports registered with `registerImports()`. These imports are no longer called by `Board::serviceloop`.

```
void unregisterImports();
```

enableImports() Method

This method allows calling the streaming imports if they were previously disabled.

```
void enableImports () ;
```

disableImports() Method

This method stops calling the streaming imports if they were previously enabled.

```
void disableImports () ;
```

isImportDisabled() Method

This method checks whether calling streaming imports is allowed or not.

```
bool isImportDisabled () ;
```

This method returns the following:

- *true*: Calling is disabled.
- *false*: Calling id enabled.

checkImports() Method

This method checks whether there is any pending message for any of the imports.

```
bool checkImports () ;
```

This method returns the following:

- *true*: there is a pending message.
- *false*: there is no pending message.

serviceLoop() Method

This method calls the pending ZEMI-3 transactor's import functions, if any. The imports requested by the software are executed.

```
void serviceLoop () ;
```

serviceLoopWithWait() Method

This method calls the ZEMI-3 transactor service loop and returns only when one import call is executed or after the specified timeout.

```
void serviceLoopWithWait (int timeout=0) ;
```

where, timeout is the value of the timeout in seconds.

startBackToBack() Method

This method enables Back-to-Back mode for the ZEMI-3 transactor.

```
void startBackToBack () ;
```

stopBackToBack() Method

This method disables Back-to-Back mode for the ZEMI-3 transactor.

```
void stopBackToBack () ;
```

isBackToBackStarted() Method

This method checks whether Back to Back mode is enabled or not.

```
void isBackToBackStarted () ;
```

This method returns the following:

- *true*: when this mode is enabled.
- *false*: when this mode is disabled.

flushBufferedPorts() Method

This method flushes import/export requests pending on the ports that are automatically buffered by the system. It ensures that all pending requests are processed.

```
void flushBufferedPorts ();
```

getXtorInfos() Method

This method returns the following information about the import, from which it is called, in the `xtorInfos` structure:

- *function*: The name of the import function.
- *scope*: Scope of the function call (for example, `xtor.ins`).
- *filename*: The name of the Verilog file in which the call is coded.
- *line*: Number of the line concerned in the file pointed by file name.

The syntax for this method is:

```
xtorInfos* getXtorInfos() ;
```

getErrorStatus() Method

This method indicates if an error is detected inside the ZEMI-3 transactor.

```
bool getErrorStatus ();
```

This method returns the following:

- *true*: an error is detected.
- *false*: no error detected.

getExportActiveStatus() Method

This method indicates whether an export task is currently running for the ZEMI-3 transactor.

```
bool getExportActiveStatus ();
```

This method returns the following:

- *true*: an export task is running.
- *false*: no export task is running.

Using a ZEMI-3 Transactor in a ZCEI Testbench

This section includes the following topics:

- Instantiating an Object
- Creating the Transactor Object
- Initializing a ZEMI-3 Transactor

Instantiating an Object

You must include in the testbench the `<my_xtor>.h` header file, generated by **zCui** in the `zebu.work` directory. This file declares your transactor interface class, which is derived from the base class `ZEMI3Xtor`. To handle the ZEMI-3 transactor, an object of this transactor class has to be instantiated in the testbench.

For example:

```
namespace ZEMI3_USER
{
  class trans : public ZEMI3Xtor
  {
  public:
    trans();
  };
}
```

Creating the Transactor Object

You must create an object for each instance of the transactor instantiated in the hardware top file. The object must be created between the calls to `ZEBU::Board::open` and `ZEBU::Board::init`.

For example:

```
// Board open
Board *z = Board::open("../zebu.work");

// Creation of the transactor objects
ZEMI3_USER::trans *t0 = new   ZEMI3_USER::trans ();
ZEMI3_USER::trans *t1 = new   ZEMI3_USER::trans ();
```

Initializing a ZEMI-3 Transactor

Use the `init()` method to initialize the transactor. This method connects the transactor's ports.

The `init()` must be called after creation of the transactor object and before `ZEBU::Board::init`. The arguments are the Board object and the transactor instance name (in the hardware top). For details about this method, see Using the ZEMI3Xtor Class.

For example:

```
// Init of the streamers
t0->init(z, "trans0");
t1->init(z, "trans1");

// Init of zebu
z->init();
```

Running a ZEMI-3 Transactor

When interfacing a ZEMI-3 transactor with a ZCEI environment, the testbench explicitly calls the export functions and transactor service-loop method services the import functions.

In a heterogeneous environment, one of the following ways is supported to call the imports:

- Using the transactor local `ZEMI3Xtor::serviceLoop()` call, which directly checks if there are messages available on the message port(s).
- Registering a callback for the import. The callback is executed during the `Board::serviceLoop()` method. The `ZEMI3Xtor::registerImports()` method does the callback registration, if required.

Thus, you can control and decide how the transactor imports should be handled.

Note:

These two modes cannot be used simultaneously for the same transactor.

Using the Local Transactor serviceLoop

When called, `ZEMI3Xtor::serviceLoop` checks if an import is called from the hardware part of the transactor. If that is the case, it calls the corresponding user import function. The advantage of this method is that it can be called in a separate thread serving only a specific transactor import. When using an application with one thread for each transactor, `ZEMI3Xtor::serviceLoop` should be used.

For example:

In the following example, there are two transactors and one thread is created for each of them.

```
bool TEST_END = false;

// Thread function: Server of import
void xtorImportServer(ZEMI3Xtor *xtor)
{
  while(!TEST_END) xtor->serviceLoop();
}

int main()
{
  Board *z = Board::open();
  // Initialization: two instances of the same my_zemi3_xtor transactor
  ZEMI3Xtor *my_handle0 = new my_zemi3_xtor();
  ZEMI3Xtor *my_handle1 = new my_zemi3_xtor();
  my_handle0->init(z, "my_zemi3_xtor_0");
  my_handle1->init(z, "my_zemi3_xtor_1");
 // Initialization of the zcei transactor
  ZCEI_xtor *my_zcei_xtor_0 = new ZCEI_xtor(z, "my_zcei_xtor_0");


   // ZeBu init
  z->init();
  // Launching threads for the ZEMI3 transactors
  pthread_t tid1;
  pthread_create(&tid1, NULL, (void *(*)(void *))xtorImportServer, (void
 *) my_handle0);
  pthread_t tid2;
  pthread_create(&tid2, NULL, (void *(*)(void *))xtorImportServer, (void
 *) my_handle1);
  // testbench execution
  ...
  // disconnection
  TEST_END = true;
  pthread_join(tid1);
  pthread_join(tid2);
  my_handle0->close();
  my_handle1->close();
  z->close();
}
```

Using the ZEBU::Board serviceLoop

You must register the imports of each transactor that are required to be handled by `Board::serviceLoop`. This provides some flexibility.

Here is an example indicating where the register call should be placed.

```
bool TEST_END = false;
// Thread function: Server of import
void xtorImportServer(Board *z)
{

  while(!TEST_END) z->serviceLoop();
}
int main()
{
  Board *z = Board::open();

  // Initialization: two instances of the same my_zemi3_xtor transactor
  ZEMI3Xtor * my_handle0 = new my_zemi3_xtor();
  ZEMI3Xtor * my_handle1 = new my_zemi3_xtor();
  my_handle0->init(z, "my_zemi3_xtor_0");
  my_handle0->registerImports ((int)my_handle0);
  my_handle1->init(z, "zemi3_xtor_1");
  my_handle1->registerImports ((int)my_handle1);
  // Initialization of the zcei transactor
  ZCEI_xtor *zcei_xtor_0 = new ZCEI_xtor(z, "zcei_xtor_0");
  // ZeBu init
  z->init();
  // Launching board thread
  pthread_t tid;
  pthread_create(&tid, NULL, (void *(*)(void *))xtorImportServer, (void
 *)z);
  // testbench execution
  ...
  // disconnection
  TEST_END = true;
  pthread_join(tid);
  my_handle0->close();
  my_handle1->close();
  z->close();
}
```

10

Performing Simulation of ZEMI-3 Transactors

A ZEMI-3 transactor and its standalone environment can be simulated with any SystemVerilog-compliant simulator such as VCS. The steps are as follows:

1. Writing a Top-Level Wrapper
2. Using a Software Entry Point
3. Launching the Simulation

These steps are described in the following subsections.

Writing a Top-Level Wrapper

The connection between the clock ports, the DUT, and the transactor is performed through the same hardware top file.

For example:

```
module hw_top();
  wire [7:0]   mem0_addr;
  wire [31:0]  mem0_din;
  wire [31:0]  mem0_dout;
  wire         mem0_we;
  wire [7:0]   mem1_addr;
  wire [31:0]  mem1_din;
  wire [31:0]  mem1_dout;
  wire         mem1_we;
  reg clk;
  initial clk <= 0;
  always #10 clk <= !clk;
  dut dut0(

      .clk(clk),
     .mem0_addr(mem0_addr),
     .mem0_din(mem0_din),
     .mem0_dout(mem0_dout),
     .mem0_we(mem0_we),
     .mem1_addr(mem1_addr),
     .mem1_din(mem1_din),
     .mem1_dout(mem1_dout),
     .mem1_we(mem1_we)
```

```
    );
  mem_xtor mem_xtor_0  (.clk(clk),
                        .mem0_addr(mem0_addr),
                        .mem0_din(mem0_din),
                        .mem0_dout(mem0_dout),
                        .mem0_we(mem0_we),
                        .mem1_addr(mem1_addr),
                        .mem1_din(mem1_din),
                        .mem1_dout(mem1_dout),
                        .mem1_we(mem1_we)
                      );
endmodule
```

Using a Software Entry Point

The testbench function is not available in simulation as it is in emulation.

To create a testbench function in simulation, you can add the following code to the hardware part of the transactor:

```
`ifdef SIMULATION
  import "DPI-C" context task testbench();
  initial begin
    testbench();
    $finish;
  end
`endif
```

Launching the Simulation

The file names used in this section match the memory transactor example described in Writing a Top-Level Wrapper.

To launch the simulation using VCS, perform the following steps:

1. Create a `mem_xtor.h` header file that includes the prototypes of the export functions:

   ```
   #include <svdpi.h>
   extern "C" export (...);
   #define XTOR_SCOPE "hw_top.mem_xtor_0"
   ```

2. Compile the transactor for the HDL simulator, proceed as follows:

   ```
   $ vcs ../src/*.sv ../src/*.v ../src/tb.cc -sverilog -R
   ```

3. Run the simulation, proceed as follows:

   ```
   $ ./simv
   ```