ZeBu™
ZEMI-3 Manual

Document revision – b –

October 2009

Version 4_3_3 / 6_1_1

Copyright Notice
Proprietary Information

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

Figures

Tables

About this Manual

Overview

This manual introduces the ZEMI-3 infrastructure together with the advantages of transaction-based verification. It presents ZEMI-3 features and gives elements to choose the most appropriate architecture for your transactor.

Recommendations to write the HW and SW parts of your transactor are given before presenting the compilation and runtime environments for ZEMI-3 transactors.

The Tutorial chapter includes several complete transactor examples with usage.

History

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

Doc Revision	Product Version	Date	Evolution
b	4_3_3 6_1_1	Oct 09	New sections: • Basic advantage of the ZEMI-3 architecture (§1.1.3) • Advanced features (§1.2.4) • Simulation time consumption (§2.3.1) • Mixing imports and exports (§2.3.4) • Serializing calls (§2.4.4) • Lossy calls (§2.4.5) • Port buffering (§2.4.6) • Multi-cycle messaging (§2.4.7) • Optimizing transaction compilation via **`zCui`** (§2.4.8) • Activating the profiling feature (§5.4.4) • Tutorial architecture (§7.1) Updated sections: • Mixing imports and exports (§2.3.4). The level of nesting is still 2 but the definition is different • Advanced properties (§3.1.4). New advanced properties: `zemi3_highpriority, zemi3_serialize, zemi3_call_skipped, zemi3_bufferedport` • Compiling a ZEMI 3 transactor with **`zCui`** (§4.3). Specifying a Decicion Mode via DPI Advanced Settings. • **`zEmiRun`** requirements (§5.2). New overrides: `LDFLAGS, CXXFLAGS, ZEMI3_LIB, XTOR_LIB` • **`zEmiRun`** options (§5.4.2). Debug option removed (`-d`). New options added (`-b, -i, -l,-r, -t, -u`). • Running a ZEMI-3 transactor (§6.2.4). Warning added: import calls mechanisms are no longer ON by default.
a	4_3_2	Apr 08	First Edition

Typographic conventions used in this manual

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

Input file contents, such as code examples, scripts or driver declarations are shown in mono-space "Courier New" font with left and bottom borders.
For example, the following line of a script describes a memory command that accepts a user-declared variable and a user-selected option:

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

Where:

<port-name>	User-declared variables are inside angled brackets ("<" and ">"), such as a port name in the present example. Do not type angled brackets when changing parameters to user names.
[rw-mode]	Options are presented inside square brackets. The available options will be listed and described in the text following the syntax rules. Do not type square brackets when changing options to user names.

Example:

```
memory set_rw_mode myPort readbeforewrite
```

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

```
$ zBuild <script_file_name>
```

Parameters and options are displayed in the same way as for input files contents: parameters inside angles brackets and options inside square brackets.
The shell prompt in this manual is $ for user environment and # for supervisor environment.

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

```
##############################################
# Copyright (c) 2002-2008                    #
# Emulation and Verification Engineering SA  #
#--------------------------------------------#
# <Tool Name>                                #
# revision :                                 #
# date :                                     #
#--------------------------------------------#
##############################################
```

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

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

1 Introduction

1.1 Using transactors in the ZeBu environment

Transaction-based verification changes the level of abstraction at which your testbenches are written. Instead of writing testbenches based on signal assignments and signal monitoring, the testbenches are written with high-level commands or actions (such as *read something* or *get something*). Using these high-level commands, testbenches are easier to write and faster to create.

A transactor is a behavioral model that gives a higher representation of an interface. This interface can be simple or complex, standard or proprietary. For each behavior, a transactor can be written once and reused as many times as required in one or more projects.

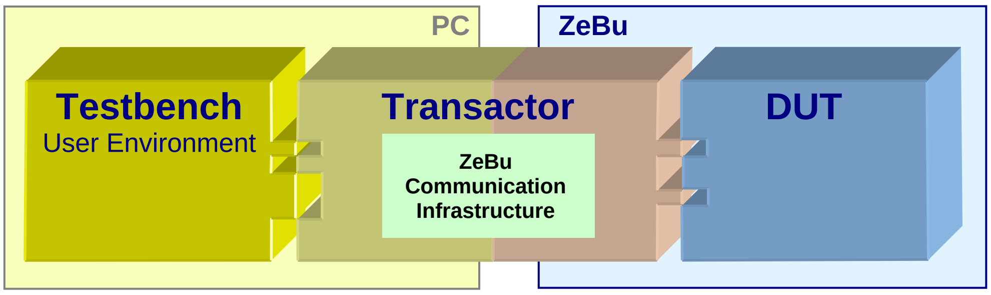


Figure 1: Transaction-based verification environment

Transactors are useful for functional testing of designs, in particular for streaming operations or connecting the design to a complex interface such as PCI Express.

Several transactors can be combined to create an appropriate environment when the DUT has several high-level interfaces, as can be the case in a multi-media SoC (e.g. one transactor for the digital video and another one for the USB interface).

A ZeBu transactor combines SW and HW to drive the DUT running on ZeBu:

- The HW part is written at a synthesizable level and runs on ZeBu.
- The SW part is usually written in C, C++ or SystemC and runs on the PC.

Each part manipulates the data at its own level:

- The SW part manipulates data at a transaction level and does not consider the signal level. It is also in charge of sending data to the HW part in an appropriate form.
- The HW part is similar yet simpler than a BFM (Bus Functional Model) described in Verilog. It mainly focuses on writing and reading data.

Example:
In a memory transactor, the SW part processes high-level commands such as requests to read or write data and the HW part manages signal-level accesses to the DUT (`data`, `address`, `we`, `ce`).

1.1.1 Off-the-shelf ZeBu transactors

EVE has developed various ZeBu transactors for the most usual communication standards (PCIe, USB, JTAG, display, Ethernet, RS232, AMBA, etc.) and applications (LCD virtual screen, RS232 console, SRAM-type memory exceeding 1GB, audio, video, etc.)

Each transactor is available as a complete package which includes the SW and HW parts and technical documentation for an efficient integration with your DUT and your verification environment.

1.1.2 ZeBu solutions to develop transactors

You can also develop transactors for ZeBu on your own in two different environments:

- **zcei environment**: the HW part of the transactor is described in RTL and communication between HW and SW parts uses a handshake protocol that must be created manually.
- **ZEMI-3 environment**: the HW part uses behavioral code and communication between HW and SW parts is defined at a message level, through function calls.

1.1.3 Basic advantage of the ZEMI-3 infrastructure

If zcei environment provides tight control over communication and allows the best possible speed, ZEMI-3 infrastructure speeds up the creation of complex transaction-based environments for design verification at a small performance price.

1.2 Designing transactors with the ZEMI-3 infrastructure

1.2.1 Introduction to ZEMI-3

ZEMI-3 main goal is to allow easy writing of transactors that are required for functional testing of a design.

ZEMI-3 is based on two concepts:

- Communication between HW and SW parts uses inter-language function calls, based on the Direct Programming Interface (DPI) mechanism, which is part of the IEEE standard for SystemVerilog (IEEE 1800-2005).
- The description of the HW part of the transactor uses behavioral coding style.

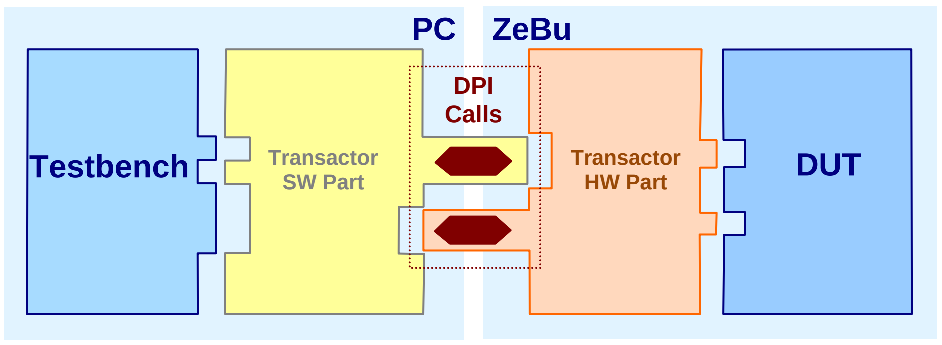


Figure 2: Architecture of a ZEMI-3 transactor

In the ZEMI-3 infrastructure, signal-level communication between HW and SW parts of the transactor (through the ZeBu handshake protocol) is hidden from the user. The higher level of communication also improves the legibility of the source code.

The ZEMI-3 infrastructure balances runtime performance and easy environment creation. Advanced modes are also available to improve the performance of ZEMI-3 based environments (see Section 1.2.4).

1.2.2 SystemVerilog DPI

1.2.2.1 Introduction

In order to build a heterogeneous environment with models described at different abstraction levels, the SystemVerilog DPI defines the way HW and SW models interact through function calls which can be done either from SW-to-HW or from HW-to-SW.

For detailed information about the SystemVerilog DPI, refer to Section 26 of the IEEE standard for SystemVerilog (IEEE 1800-2005).

1.2.2.2 Inter-language function calls

The SystemVerilog DPI describes different ways to declare functions and tasks between HW and SW parts of the environment:

• **Import functions**	C functions called from SystemVerilog
• **Export functions**	SystemVerilog functions called from C
• **Export tasks**	SystemVerilog tasks called from C

Note: When using the SystemVerilog DPI, the difference between functions and tasks is the same as in the Verilog standard (IEEE 1364), in particular concerning the capability for tasks to consume simulation time (see Section 2.3.1).

1.2.3 Using behavioral code for the HW part of a transactor

The HW part of a ZEMI-3 transactor can be described with behavioral code, which makes the description of the functional model easier since it can be done from the waveforms without the need to build a state machine.

The constraints are lower compared to coding the equivalent behavior in RTL, due to the following possibilities:

- Events can be located in `always` blocks or in other blocks.
- Clocks, edges and other events can be mixed in the same `wait` statement or in the same process. Examples of clock management in a ZEMI-3 transactor are shown in the Tutorial, in Chapter 7 of this manual.
- Signals can be assigned in several processes, allowing a modular approach with smaller, dedicated processes.
- Unbounded loops can be used.

Moreover, behavioral code improves the legibility of the code, in particular for people with a SW background. With behavioral code, the modularity of the SW is improved, making it easier to add new features to the HW part of the transactor.

1.2.4 Advanced features

1.2.4.1 Streaming mode

This mode improves the performance by optimizing single-direction data exchanges. Refer to Sections 2.3.8 and 3.1.4.1 for more details.

1.2.4.2 Protocol mode

This mode defines a set of mutually exclusive functions for a modular description of data exchanges, in particular for protocol operations. Refer to Sections 3.2.5 for more details.

1.2.4.3 Back-to-back mode

This mode provides a debug-oriented environment which can be controlled with a single-cycle precision. This mode is optimized for a scenario-based debugging and comes in addition to the default mode which has a behavior similar to the actual environment of the design. Refer to Sections 2.4.1 and 3.2.5 for more details.

1.2.4.4 Other modes

Other modes are available in the following sections:

- Prefetch calls (§2.4.3)
- Serialization calls (§2.4.4)
- Lossy calls (§2.4.5)
- Port buffering (§2.4.6)

2 Choosing a Transactor Architecture

2.1 Introduction

The transactor organizes the interaction between the testbench and the DUT:

- The SW part of the transactor is written in C/C++ and interacts with the user testbench through high-level commands.
- The HW part interacts with the DUT at a cycle-/signal-level to process multi-cycle operations as an untimed functional model. These interfaces make the integration of the testbench and DUT easy.

In a ZEMI-3 transactor, the HW and the SW interact through inter-language function calls. The transaction data corresponds to the arguments of the function call, and all data transfers and time domain synchronizations happen automatically and transparently. The calls can be initiated either by HW or SW according to your verification environment.

Choosing the architecture of the transactor consists in defining which operations are processed by the SW part of the transactor and which are processed by the HW part.

2.2 Deciding for the best transactor architecture

2.2.1 Do the processing where it makes sense

A transactor is composed of HW and SW parts and some operations fit better in one part than the other. For example, complex arithmetic operations are more efficient in SW but they cost a lot of logic. On the opposite, bit-level operations such as CRC calculations are usually more efficient in HW.

2.2.2 Avoid using a large bandwidth if not required

Using ZEMI-3, it is quite simple to stream data out of the DUT for debugging purposes. However, streaming thousands of bits at every cycle is not an efficient way of using ZeBu and would result in a performance drop as well as potential compilation issues. It is better to consider writing a small piece of code in the HW which checks the data and raises flags on the events you need to follow.

When required to stream data to the SW, consider sending only useful data.

2.2.3 Use multi-cycle transactions

The maximum performance of the ZEMI-3 infrastructure is obtained when multiple cycles are run between 2 communications between the SW and HW parts of the transactor. The more cycles in a single transaction, the better it is.

The ratio between the number of clock cycles and the number of inter-language calls should be maximized for better performance of your verification environment. Examples of clock management in a ZEMI-3 transactor are shown in the Tutorial, in Chapter 7 of this manual.

2.3 Data exchange between HW and SW

The calls can be initiated either by HW or by SW according to your verification environment. The choice for HW- or SW-initiated functions depends on the architecture of the transactor.

2.3.1 Simulation time consumption

The *simulation time consumption* concept used in the following sections is based on the fact that tasks do consume simulation time (i.e. clock cycles) when they wait for events or clock edges to occur.

For instance, when using the SystemVerilog DPI, the difference between functions and tasks is the same as in the Verilog standard (IEEE 1364), in particular concerning the capability of tasks to *consume time* while functions must return their output without consuming time.

2.3.2 HW-initiated transactions

HW-initiated transactions are preferred to transfer data from HW to SW, in particular for a HW-to-SW streaming application or when the HW needs a complex operation result which is easier to implement in the SW part.

For that purpose, you use an import function which is a C-language function called from the SystemVerilog code, with optional inputs coming from and outputs returning to the HW part of the transactor. It allows the HW to send some data to the SW and then use the results processed by the SW part.

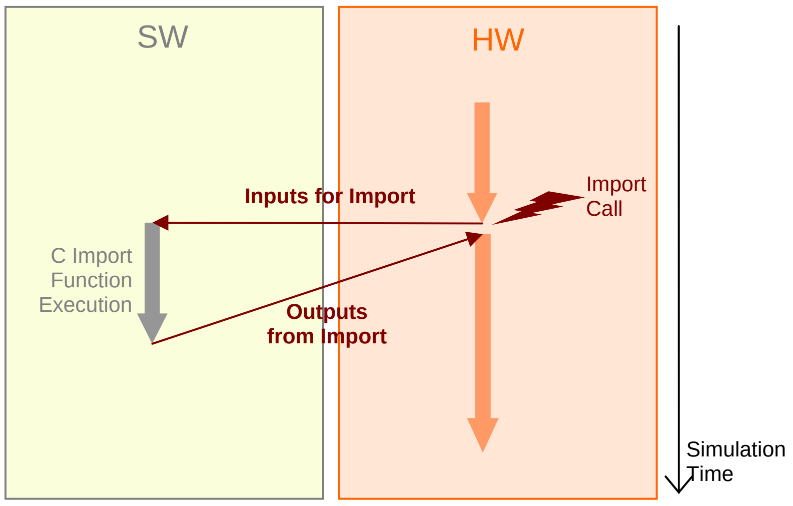


Figure 3: HW-initiated transaction

During the execution of the import in the SW, the HW part of the transactor is stopped (the design clocks do not run). Such calls strongly impact the performance if they are used at every DUT clock cycle.

Note: The C import function is executed in a thread.

2.3.3 SW-initiated transactions

SW-initiated transactions are preferred to transfer data from SW to HW, in particular for a SW-to-HW streaming application. For that purpose, you use an export function which is a SystemVerilog function (or task) for which inputs come from the SW part of the transactor. It allows the SW to send some data to the HW and then use the results processed by the HW part.

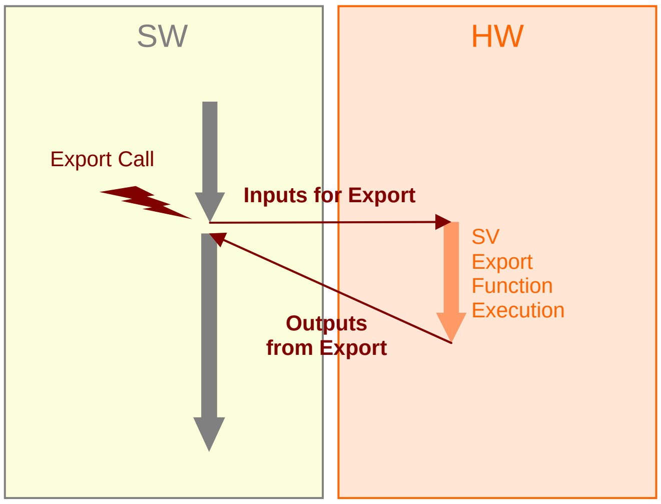


Figure 4: SW-initiated transactions

During export function execution, the SW side of the transactor waits for the outputs.

2.3.4 Mixing imports and exports

In a ZEMI-3 transactor, an export function can be called from an import function and an import function can be called from an export function/task. The maximum level of nesting is 2, i.e. one import can call several exports in succession BUT when an import calls an export, then that export cannot call another import.

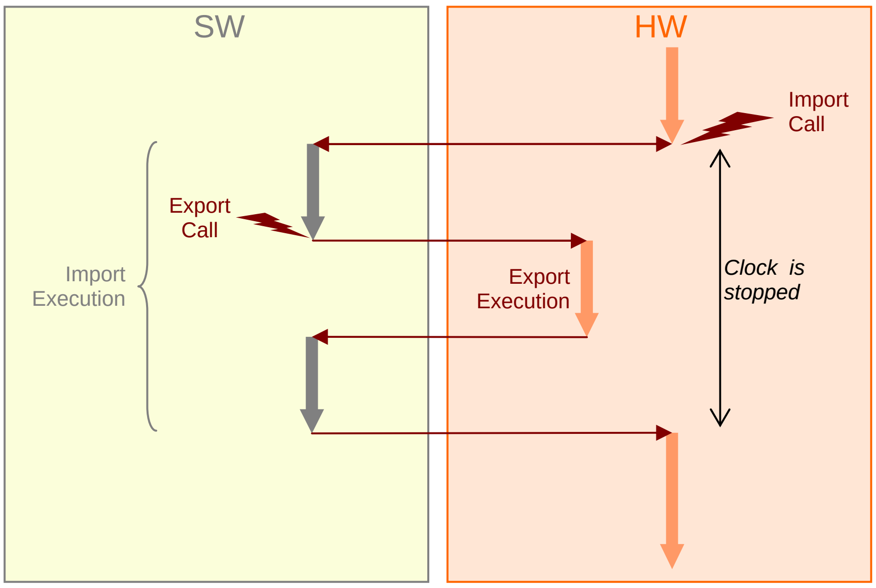


Figure 5: Calling an export from an import (context import)

An import function which calls an export must be declared as a context import, as described in Section 3.1.1.1. The export which is called from the import must not consume DUT clock cycles because **ZEMI-3 imposes that import functions do not consume simulation time**.

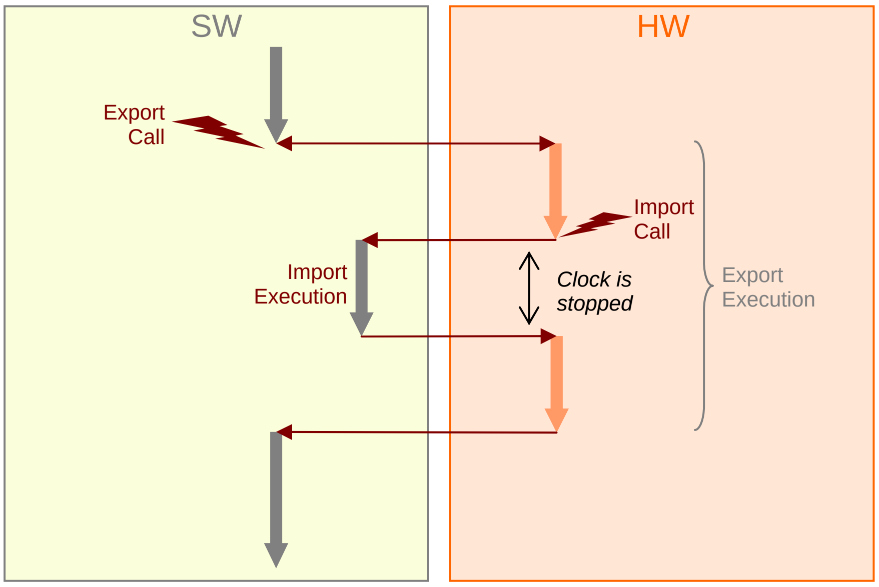


Figure 6: Calling an import from an export

The export can consume some simulation time but the included import cannot.

2.3.5 Functions and tasks

For import calls, only functions can be used since the SW part does not directly control simulation time on the HW side. Functions execute immediately from a HW point of view, which means that any call of an import function stalls the clock of the transactor.

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 immediately.

Export tasks and functions are triggered by SW calls. They execute in parallel of all other HW blocks. At most, one instance of an export call can be active at one time. If there are multiple calls, then they are "queued" (in particular, this affects export tasks in that the second task will be delayed until the first task has finished.)

Clocks are controlled in tasks directly by referring them through `wait` statements on their positive or negative edges, as in the following example:

```
always
begin
  @(posedge clk)
  a <= b;
  @(negedge clk)
  a <= c;
end
```

Note: A task without a `wait` statement is equivalent to a function.

From a user point of view, arguments of functions and tasks are exchanged between HW and SW sides without any delay at the synchronization points (see Section 2.3.7). The ZEMI-3 infrastructure prevents the channels from overloading.

Import tasks are not supported in the ZEMI-3 infrastructure, therefore no simulation time can be consumed in an import; i.e. only an export function (not an export task) can be called from an import in case of nesting.

The size of the data exchanged between HW and SW at a synchronization point can be up to 8,000 bits in each direction (input or output). If the data are bigger than this limit, then the messages will be automatically turned into multi-cycle messages.

2.3.6 Managing the call order of SW import functions

When calls are done in the same process (`always` or `initial` block), the order of calls is guaranteed the same on both HW and SW sides, even in a streaming environment.

When calls are done in parallel in multiple processes, the order in which they happen on the SW side can differ from the order in which they were actually called from the HW side. This order can become deterministic when the *serializing calls* advanced feature is used (see Section 2.4.4).

2.3.7 Synchronization points

Synchronization points exist when bidirectional data exchanges occur between the HW and SW parts of the transactor. In such a case, the two parts need to wait for each other.

Whenever possible, it is recommended to avoid functions or tasks mixing inputs and outputs in order to minimize such synchronization points. For example:

- Using an export function or task with outputs (or with a return code) prevents the SW from performing any other operation in parallel.
- Using a non-streaming import function in a clocked `always` block prevents the HW clock from progressing (see details on streaming data in Section 2.3.8). If possible, replace the import function by a streaming import function and a streaming export function.

If such a case is required, make sure that it is done rarely or that it corresponds to a large number of cycles of activity in the design.

Synchronization points are obviously required to create the full environment but there must be as few of them as possible because they may impact the performance of the system. Using too many synchronization points may ultimately result in every-cycle communication and thus give the worst performance. The best performance can be achieved when those points are done after large number of cycles, or even avoided as in data streaming.

2.3.8 Streaming data

It is interesting to remove the synchronization points in a pipe-like channel communication when data moves in a single direction, from SW to HW or from HW to SW. Such situations exist when import or export functions have only inputs. Functions with only outputs cannot be considered for streaming because they require bi-directional exchanges: one direction when the processing is initiated although there is no input; the other direction when the outputs are sent.

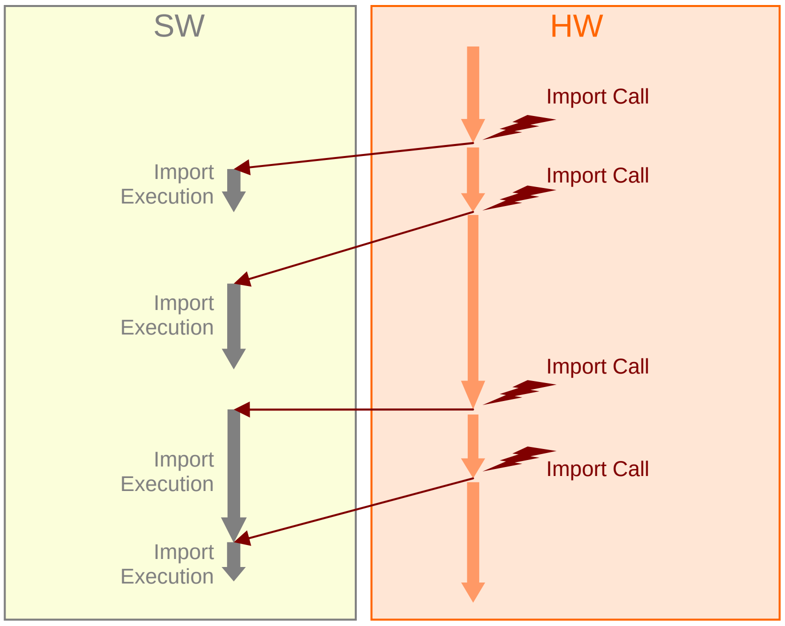


Figure 7: Streaming import

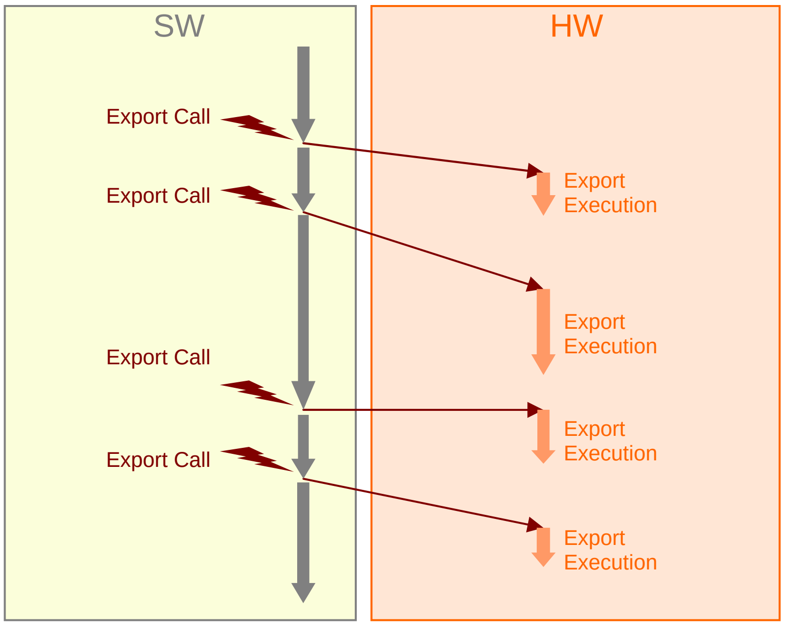


Figure 8: Streaming export

Each time it is possible, ZEMI-3 turns function calls into a streaming communication to avoid synchronization points and to optimize transactor performance. When there are only inputs, HW and SW parts can go on processing without having to wait for the other part.

In *streaming mode*, ZEMI-3 infrastructure automatically optimizes the communication speed by buffering the requests and manages the 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 *streaming mode* can be disabled for the call types which are automatically converted to streaming and context import functions can be forced into *streaming mode*, as described in Section 3.1.4.1. However, nested import and export functions, as described in Section 2.3.4, must never be forced into *streaming mode* because it would cause unexpected runtime behavior.

It is recommended to create transactions in *streaming mode* as often as possible because ZEMI-3 streaming optimization cannot be done when synchronization points are used.

If output-only import functions are heavily used in the environment, the *prefetch* mechanism improves performance, as described in Section 2.4.3.

2.4 Advanced features

All implementation details for the following features can be found in Section 3.1.4

2.4.1 Back-to-back mode

When verifying your design with a transactor, you may be interested in two different debugging strategies:

- System functional testing: in *Real Traffic* mode, transactor clocks are always running. The delay between consecutive transactions is not deterministic.
- Corner-case debugging: 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.*

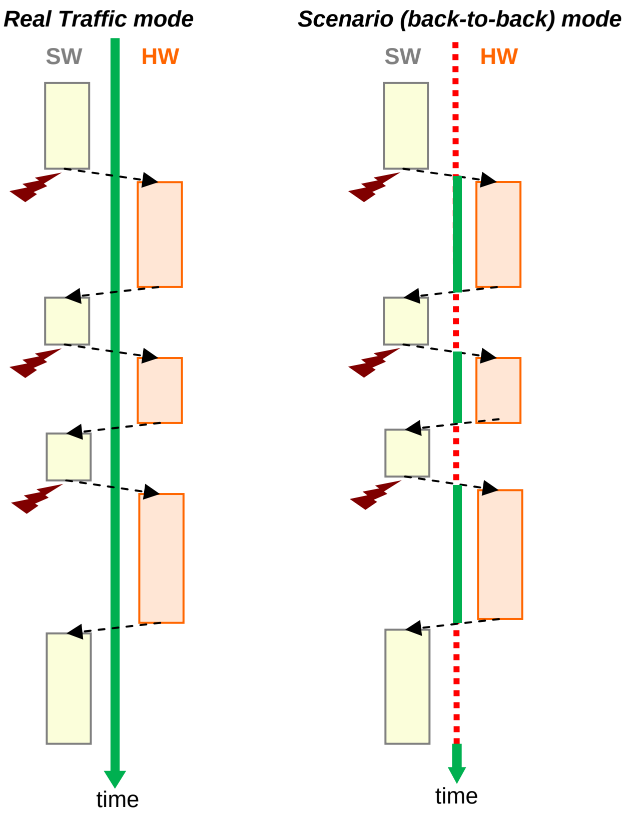


Figure 9: Back-to-back mode

During your verification process, you may need to switch from real traffic to scenario many times, with a minimum development cost. Thanks to the *back-to-back mode*, you design your transactor as compliant with your real-traffic specifications and the same transactor can be operated with the scenario mode according to your debugging requirements.

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 (from **`zCui`**).

2.4.2 Modular transactor architecture

When implementing a complex protocol such as PCI Express, it may be interesting to create a modular architecture with separate SW-initiated commands such as `Read PCI` and `Write PCI`. Such architecture makes it easier to write transactor SW thanks to improved modularity. It also makes it easier to upgrade a transactor: to add a new feature, you can create a separate function without impacting the existing source code.

This architecture is known as the *protocol mode* in the ZEMI-3 infrastructure. It consists in the declaration of an additional advanced property when the export functions are declared in the SystemVerilog code.

The DUT interface signals written by protocol-gathered tasks are managed in a non-concurrent way and the scheduling of the tasks is automated.

The *streaming mode* can be used for protocol-gathered export tasks and functions.

2.4.3 Prefetch calls

When an import function has only outputs, the SW can prepare these data so that the HW part does not have to wait for data at the synchronization points. This is known as *prefetch* calls:

- Without *prefetching*, a request is sent by the HW to the SW to get new data when reaching the import call. This request is served by the SW and data are sent back to the HW. The clocks can then progress again until the next incoming data.
- For an import with input data only, there is no need to wait for the actual call to prepare the data for the next request. This operation is called *prefetching*.

Prefetching can be compared to the ZEMI-3 *streaming mode* for SW-to-HW communication, but where the SW controls the communication flow.

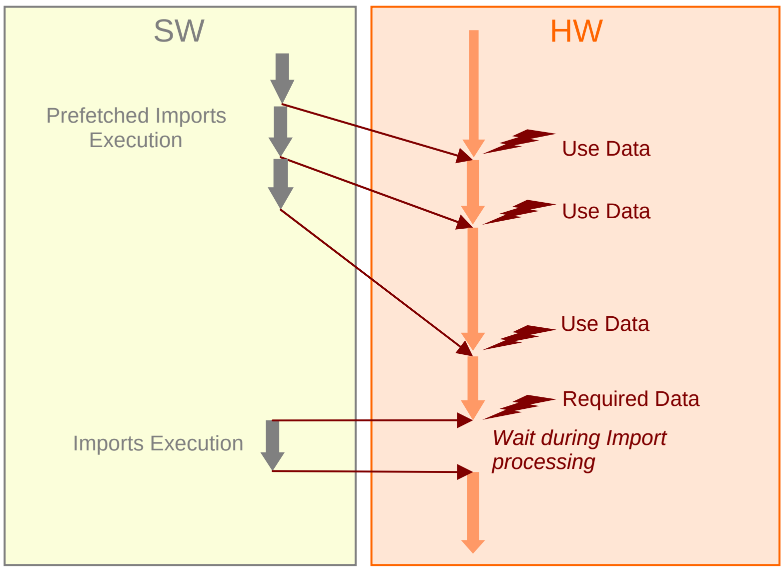


Figure 10: Prefetch mode

For that purpose, the compilation process automatically creates a specific *prefetch* function for each import function which has only outputs. The SW calls the *prefetch* function and the results of the import processing are stored until they are used by the HW part. The HW part uses the data when it needs them.

The *prefetch* mode can be enabled/disabled:

- When enabled, the original import function will not be called but the data will explicitly be sent by the *prefetch* function.
- When disabled, the original import function will be called.

The system will accept some sets of data, but may refuse new ones if all levels of buffering are full.

2.4.4 Serializing calls

When two import calls come from different processes, ZEMI-3 does not guarantee that the SW handles multiple calls in the same order as the HW does.

This is quite counter-intuitive when the two calls are also separated in time. In that case, a call that is generated later in time may be seen first by the SW.

When it is important that the order of calls be consistent, the calls should be explicitly serialized by the user.

One guideline is that if the calls operate on the same data in the SW, then they should be serialized (e.g. read memory and write memory calls that operate on the same memory implemented in SW).

Serializing calls allow you, at implementation level, to use the same HW port for several imports. However if an import requires a big throughput, then you should not serialize this import.

2.4.5 Lossy calls

This property can be attached to an import to keep it from sending messages to the SW when the communication channel is too busy. See more details in Section 3.1.4.4.

2.4.6 Port buffering

The *port buffering* property can be attached to a streaming import to create buffers for the message. Using this property can be useful when using small messages which do not require very small latency for monitoring. See more details in Section 3.1.4.5.

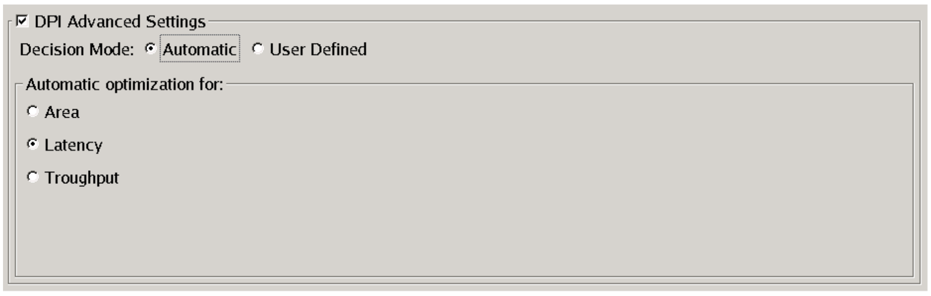
2.4.7 Multi-cycle messaging (port multiplexing)

This feature allows you to split messages into packets before they are sent. A smaller HW port would then be necessary and area will be saved in the FPGA. This feature applies to imports and exports, and it is available from **`zCui`**.

2.4.8 Optimizing transaction compilation via `zCui`

You can decide, via **`zCui`**, to automate the optimization of transactor compilations or to specify your own user-defined settings.

2.4.8.1 Automatic optimization of transaction compilation



☑ DPI Advanced Settings
Decision Mode: [✓ Automatic] User Defined
Automatic optimization for:
Area
[✓ Latency]
Troughput

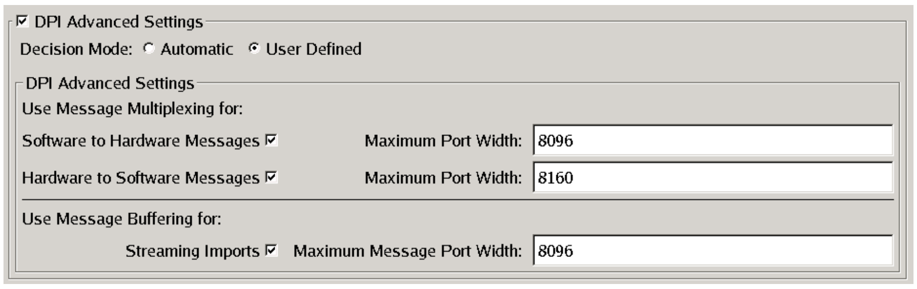
Figure 11: Automatic optimization of transaction compilation

Optimizing the area:
Use the **Area** option to factorize the usage of HW ports for imports and exports. Using this option also reduces, whenever it is feasible, the message size by splitting the message in sequential packets (multi-cycle).

Optimizing the latency:
Use the **Latency** option to minimize the exchange time between HW and SW and vice versa.

Optimizing the throughput:
Use t**he Throughput** option to increase the bandwidth by bufferizing streaming import messages. Note that you can use the `zemi3_highpriority` pragma (§3.1.4) when a transactor contains an import which needs maximum throughput.

2.4.8.2 User-defined optimization of transaction compilation



☑ DPI Advanced Settings
Decision Mode: Automatic [✓ User Defined]
DPI Advanced Settings
Use Message Multiplexing for:
Software to Hardware Messages ☑ Maximum Port Width: 8096
Hardware to Software Messages ☑ Maximum Port Width: 8160
Use Message Buffering for:
Streaming Imports ☑ Maximum Message Port Width: 8096

Figure 12: User-defined optimization of transaction compilation

Message multiplexing:
Select this option to specify message multiplexing for SW-to-HW and/or HW-to-SW communication. Indicate the maximum port width in the associated field (default values are 8096 for SW-to-HW and 8160 for HW-to-SW communication).

Message buffering:
Select this option to specify streaming imports. Indicate the maximum message port width in the associated field (default is 8096).

3 Writing a ZEMI-3 Transactor

3.1 Writing the HW part

3.1.1 Imports and exports in the SystemVerilog code

3.1.1.1 Import function

An import function is declared in the SystemVerilog code so that it can be called later in the description of the HW part of the transactor. Both declaration and call must be done in the same Verilog module.

The implementation of the function is in the C source code of the SW part of the transactor, as described in Section 3.2.2.

Example:

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

// Import call in the SystemVerilog code
reg addr;
reg data;
always @(posedge clk)
begin
  addr <= addr + 1;
  read_addr(addr, data);
  val <= data;
end
```

When the user wants to use the additional scope information, the import function must explicitly be declared as a context import. This is required, in particular, when the import function is nested within an export function.

Example of declaration of a context import in the SystemVerilog code:

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

3.1.1.2 Export function/task

In case of an export, the SystemVerilog code includes the declaration and the implementation of the task or function. Both declaration and implementation must be done in the same Verilog module.

The declaration of the export function does not include the interface of the function. Parameters of the function are declared only in the module where it is implemented.

Example of an export declaration and call:

```
// 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
```

3.1.1.3 Prefer blocking assignment in export tasks

Since ZEMI-3 is based on the IEEE standard for SystemVerilog, non-blocking assignments in export tasks can end in unexpected runtime behavior, in particular when assigning a value to an output argument as the final statement of the task.

Export task with blocking assignment 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
```

Using non-blocking assignment of output arguments results in never getting the value back to the SW because the non-blocking assignment is not executed right-away. Using a blocking assignment in export tasks avoids such situations.

3.1.2 Using behavioral coding style

The HW part of a transactor designed with ZEMI-3 can be described using behavioral code.

3.1.2.1 Supported behavioral subset

The following elements are supported in the behavioral description of the HW part of the transactor:

Table 1: Supported sub-set of behavioral code

Supported Elements	Example
Standard RTL subset	
User-Defined Primitives (UDPs)	Typically Xilinx cells.
`initial` statements	`initial counter = 0;`

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

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

Supported Elements	Example
Multiple clocks/edges in 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 combination	`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`
SystemVerilog data types	`logic`, `bit`, `int`, `longint`, `byte`, `void`
Block disable within a process	
Delta delays for races	`zemi3_event` advanced property

3.1.2.2 Limitations

The following features are not supported in ZEMI-3:

- SystemVerilog data types other than `bit` and `logic`
- `fork/join`
- `cmos`, `nmos`, `pmos`, `rcmos`, `rnmos`, `rpmos`
- Strengths on drivers
- Modulo and non power of 2 division
- Delays for behavior
- System tasks/functions
- Procedural `assign/deassign`
- Inter-process block disable

3.1.2.3 Clocks, multiple clocks and multiple edges

The clocks of a transactor must be declared for the compilation of its HW part, using the DVE file. Any combination of clocks and edges is supported in the transactor. It is even possible to mix in the same process multiple clocks and multiple edges of the same clock. This operation is usually done by inserting statements like `@(posedge clock)` or `@(negedge clock)` in the blocks. Any number of them can be used in the same block.

Note: From a ZEMI-3 point of view, any signal used in a @ statement is a clock which sees two types of clocks:

- Controlled clocks (clocks defined as such in **`zCui`**)
- Sampled clocks (any other clock)

However, you should not mix controlled clocks and sampled clocks in the same event expression. When targeting performance, it is advised to limit the number of clocks used in a transactor. If possible, divided clocks and gated clocks should be avoided.

Clocks can be used in export tasks without specific limitation for the combination of clocks and edges. It is possible to describe a sequence of actions to be performed on a set of signals and to call it from the SW.

Examples of clock management in a ZEMI-3 transactor are shown in Chapter 7.

Warning: Using sampled clocks slows down the transactor speed.

- Reason: It is due to the fact that edge detection on a sampled clock means stopping the controlled clocks to get consistent values from the registers being clocked on controlled clocks or coming from the design.
- Solution: If your transactor uses no sampled clock to send messages to the SW, disable the stopping of controlled clocks by adding an attribute at compilation time. This attribute can only be used in the last phase (speed optimization) of transactor development, i.e. when the transactor is working fine.

In the example below, data is controlled by a controlled clock and is sent to the SW on a sampled clock edge. **In this case, the optimization attribute must not be used**.

```
module xtor(input ctrl_clk, input en, input sample_clk, output
[31:0] data);
reg [31:0] data;
always @(ctrl_clk)
data <= data + 1
always @(sample clk)
if (en)
send_data_to_sw(data);
endmodule
```

In the second example, optimization can be done since no sampled clock exists.

```
module xtor(input ctrl_clk, input en, output [31:0] data);
reg [31:0] data;
always @(ctrl_clk) begin
data <= data + 1
if (en)
send_data_to_sw(data);
end
```

The attribute used in **`zCui`** (see Section 4.3) to enable transactor optimization is:

```
Compile:Zxtor:AsyncStopClock=false
```

3.1.2.4 Usage of blocking and non-blocking assignments

Non-blocking should not be used for function return value or outputs since functions cannot consume time. However, use of non-blocking for side effects is allowed.

Note: Mixing blocking and non-blocking assignments in the same register is not allowed and will result in a synthesis error.

3.1.3 Inserting delta delays

In the HW part of the transactor, it is possible to add some *delta delays* clocked by the ZeBu driverClock which is a high-speed, internal clock of ZeBu. Such delays (also known as *delta-cycle delays*) can be used to balance performance and logic size because delta delays introduce sequential elements in the behavioral design. 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 help reduce the generated logic, but usually reduces the speed, in particular because multiple delta delays may be generated per design cycle.

A delta delay is added by adding a `zemi3_event` advanced property on an empty statement (omitting the semi-colon creates inappropriate runtime behavior):

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

You can also use these delta delays to split long combinational paths, or to reduce the number of ports required on a memory at a given time, 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 would require a high amount of logic if created without delta delays. The insertion of delta delays at each access transforms the memory into a single-port memory with 1024 access but the performance is lower since 1024 driverClock cycles are necessary.

3.1.4 Advanced properties

The following table lists the advanced properties which can be used in the transactor to modify or optimize its behavior:

Advanced properties	Description
`(* zemi3_event *);`	Inserts 1 delta delay (§3.1.2.4).
`(* zemi3_stream=0 *)`	Disables the *streaming mode* on import/export (§2.3.8).
`(* zemi3_highpriority *)`	Sets the ports associated with import/export to high priority.
`(* zemi3_protocol=<my_protocol> *)`	Specifies the name of protocol when exported tasks share critical resources (§2.4.2).
`(* zemi3_serialize=<my_serializer> *)`	Specifies the name of the serializer when imported tasks are serialized (§2.4.2)
`(* zemi3_call_skipped = "fail" *)`	Specifies the import to be a *lossy call* and the name of the bit indicating if the import has been performed or not
`(* zemi3_bufferedport[=<num>] *)`	Specifies the import to be buffered and gives the number of messages to bufferize

3.1.4.1 Forcing/Disabling streaming mode

If you do not want ZEMI-3 to optimize DPI calls into streaming communication, you can disable the *streaming mode* with the `zemi3_stream=0` advanced property in the DPI call declaration:

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

By default context import functions are not declared as streaming because they can call export functions. If the context import implementation contains no export call, the *streaming mode* can be enabled with the `zemi3_stream=1` advanced property in the DPI call declaration:

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

3.1.4.2 Protocol mode

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

Example:

```
(* zemi3_protocol="dut_inc_dec", zemi3_stream=1 *)
export "DPI-C" task inc_dut_input;

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

3.1.4.3 Serialization

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

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);
```

3.1.4.4 Lossy calls

The *lossy calls* feature allows you to skip the call of an import if the communication channel between the HW and the SW is busy. Without the *lossy calls* feature, the import would block the HW side until the communication channel has some bandwidth available. To know if the call has been performed, a *fake* output bit has to be added at the end of the argument list. If this bit is equal to 1 after the call, then the call has not been performed.

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

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

Example of a transactor part:

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

  reg [31:0] failure ;
  bit fail ;

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

  initial failure = 32'b0 ;

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

  /* ... */

endmodule
```

3.1.4.5 Port buffering

For streaming import calls, it is not necessary that the software sees the import call immediately as it happens. Instead, we can save the cost of sending many small messages by accumulating a number of these messages and sending them as a block.

Sending fewer messages allows you to increase the global funtionning frequency of the controlled clocks. However, the latency between the real HW call and the import function call is increased.

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

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

  The optional value specifies the size of the final accumulated block. If
not given, a default value will be used.

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

3.1.5 Clocks

A ZEMI-3 transactor must have at least one controlled clock which is generated by the ZeBu clock generator. Such clocks are input ports of the transactor and they are declared in the DVE file by instanciating a zceiClockPort, as described in Section 4.2.

These clocks can implicitly be stopped by the transactor. If a transactor has several design clocks, all these clocks are stopped as soon as one clock waits for a clock event, i.e. until a message is sent by the SW or until the SW is ready to receive the message which was sent. If there are different clock groups, all the groups they belong to are stopped.

Controlled clocks are enabled by default. During a DPI call, if no data is available from the SW or if the SW is not ready to receive data, then all controlled clocks of the transactor are stopped automatically until data is available or until the SW is ready.

In some cases, some clocks can be generated by the DUT like in a LCD design. This kind of clock is seen from the transactor as a sampled clock; it must not be declared as a controlled clock.

Note: From a ZEMI-3 point of view, any signal used in a `@` statement is a clock which sees two types of clocks: controlled clocks (clocks defined as such in **`zCui`**) and sampled clocks (any other clock).

3.2 Writing the SW part

3.2.1 Calling export functions/tasks in the C code

An export function (or task) is a SystemVerilog function which is called from the SW part of the transactor. The prototype of the export function (or task) is available in the transactor header file generated by the compilation of the HW part of the transactor. In the source code of the SW part of the transactor, the export function or task is called as any other function:

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

Note: Before calling an export task, the current scope has to be set to an appropriate value which is the instance path name of the export function in the transactor. This is done using the SystemVerilog standard function `svSetScope`.

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

3.2.1.1 Prefetch mode

In the *prefetch mode* (described in Section 2.4.3), the import function is not called with its SystemVerilog name. A specific name is generated for that matter:

```
extern bool block_prefetch_<exportname> /**/;
```

The prefetch mechanism can send data ahead of time. Prefetch functions are generated for each import having outputs only.

To enable the prefetch mode (import is not called while *prefetching* is enabled):

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

To disable the prefetch mode:

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

Prefetch function: This function will send data to the HW. The call will block if the message port buffers are full. Thus it is a good idea to call the *prefetch* function in a separate thread, so that the main processing will never get blocked.

```
extern bool block_prefetch_<export_name> (/* data */);
```

3.2.1.2 `zemi3_ExportCallIsBlocked()` macro

This macro can be used to determine whether an export function will be executed without blocking the SW side. For example, if the HW side is not ready to receive any message calling an export at this time, the macro will return `true` until the HW side 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);
//...
```

3.2.2 Implementing import functions in the C code

An import function is a C function which is called from the HW part of the transactor, as described in Section 3.1.1.1. The import function must be implemented as an `extern "C"` for correct runtime behavior:

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

Notes:

- All import functions are resolved in the global scope.
- Calls from anywhere in the HW scope will call the same import function. If the code needs to understand the call context, then the import should not be declared as pure. Then the SystemVerilog functions `svGetScope()` can be used to determine the current call's context.

3.2.3 C types

The DPI standard defines correspondence between the HW objects and their SW counterpart types. ZEMI-3 does not use all of them, but offers the main ones for efficient communication.

Table 2: Correspondence between SystemVerilog and C language types

DPI formal argument types	Corresponding types mapped to C
`byte`	`char`
`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` (with specifically defined val

DPI formal argument types	Corresponding types mapped to C
Packed one-dimensional arrays of • `bit` types and • `logic` types **Note:** packed multi-dimensional arrays are also supported, but they are presented as single dimensional arrays on the C-side.	Canonical arrays of: • `svBitVecVal` and • `svLogicVecVal`

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

3.2.4 Supported SystemVerilog C functions in ZEMI-3

The SystemVerilog standard functions supported in the ZEMI-3 environment are listed in the following table, with a reference to the examples of the Tutorial in which they are used (for more details, refer to the SystemVerilog LRM):

Table 3: Supported SystemVerilog C functions

Prototype	Description	See Example
`svScope` **`svGetScope`**`(void)`	Gets scope of called imported function	§7.2

`svScope` **`svSetScope`**
`(const svScope scope)`
	Sets scope for exported function	§7.3

`int` **`svPutUserData`**
`(const svScope scope,`
`void *userKey,`
`void *userData)`
	Associates a data and a scope	§7.5

`void` **`*svGetUserData`**
`(const svScope scope,`
`void *userKey)`
	Gets data associated to a scope	§7.5

`const char` **`*svGetNameFromScope`**
`(const svScope scope)`
	Gets scope name	§7.6

`svScope` **`svGetScopeFromName`**
`(const char *scopeName)`
	Gets scope from name	§7.2

`int` **`svGetCallerInfo`**
`(char **fileName,`
`int *lineNumber)`
	Gets line number and file name of caller	

`const char` **`*svDpiVersion`**`(void)`	Gets DPI version	

`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`	

Prototype	Description	See Example
`void svPutPartselBit (svBitVecVal *d, const svBitVecVal s, int i, int w)`	Puts a part of `svBitVecVal` in another `svBitVecVal`	

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

3.2.5 Controlling the back-to-back mode from the user testbench

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

The following functions are available for *back-to-back* control:

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

4 Compiling the HW Part of a ZEMI-3 Transactor

4.1 Introduction

The HW part of a ZEMI-3 transactor is compiled from the SystemVerilog source files and can optionally use some EDIF netlists or some **`zMem`** memory descriptions. The compilation of the HW part of a ZEMI-3 transactor is fully integrated in **`zCui`**.

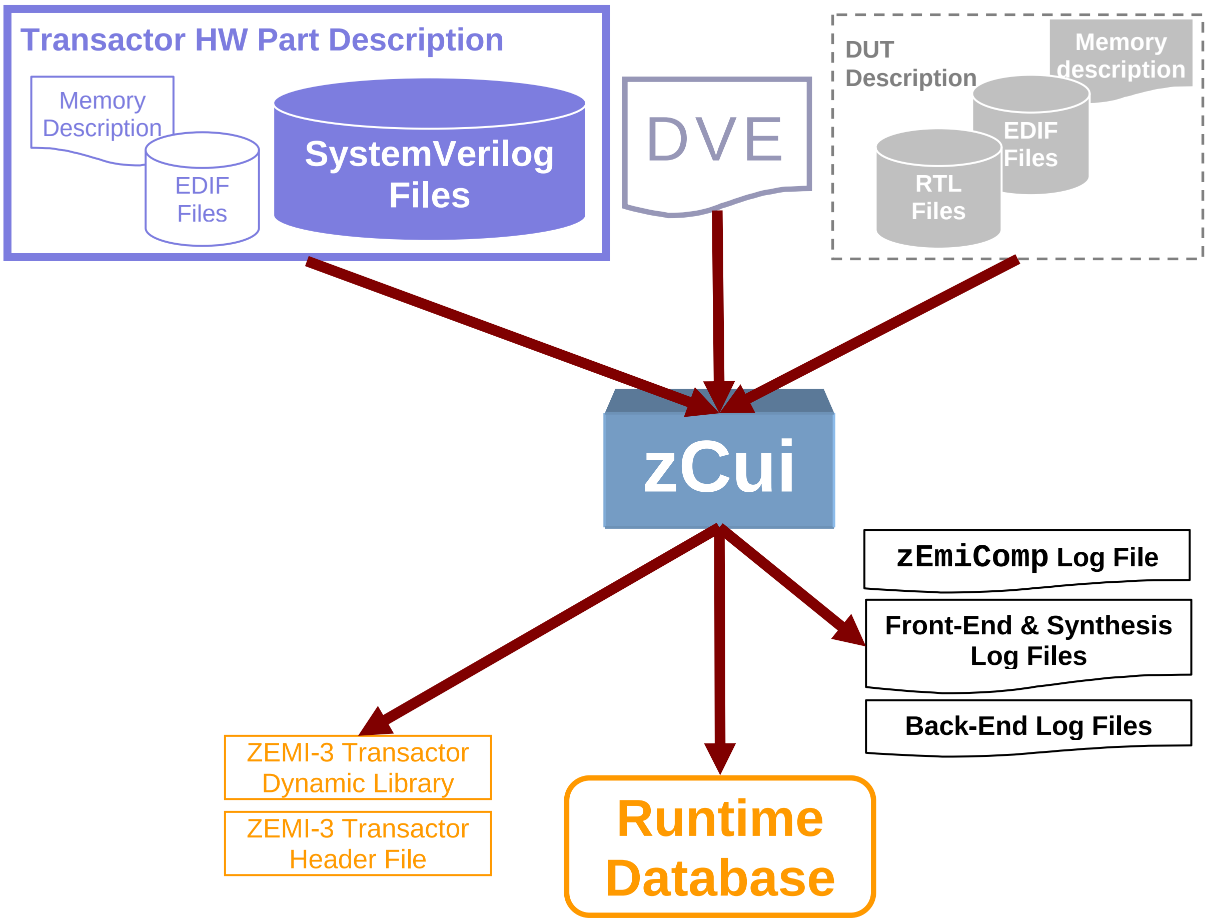


Figure 13: Compiling the HW part of a ZEMI-3 transactor

In **`zCui`**, the transactor is processed by **`zEmiComp`** in order to identify all the export and import functions and tasks. This operation is done before launching the RTL front-end for synthesis.

This compilation process requires a specific DVE file where the transactor is connected to the DUT and where the controlled clocks are declared.

The compilation of the SW part of the transactor is different according to the runtime environment. It is described later in this manual.

4.2 Instantiating a ZEMI-3 transactor in the DVE file

A ZEMI-3 transactor is instantiated in the DVE file in a similar way to other transactors and co-simulation drivers, except for the clock declaration which is slightly different.

In a ZEMI-3 transactor, the clock is declared as any other port of the transactor. Since the clock has to be a controlled clock generated by ZeBu, the clock port of the transactor is connected to the output of a zceiClockPort.

Example:

```
streamer_out s0(.clk(clk),  .data(count[31:0]) );

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

Note: Full DVE examples are available in the Tutorial in Chapter 7 of this manual.

`zCui` automatic DVE generation only applies to cycle-based environments and cannot be used for transactors. With ZEMI-3, the DVE file has to be written manually, instantiating the transactors with their connections to the DUT mapped in the design FPGAs of ZeBu.

4.3 Compiling a ZEMI-3 transactor with `zCui`

Assuming you have all the RTL source files of your transactor and DUT, you can now compile with **`zCui.`**

To compile your DUT, you should follow the usual procedures which are out of the scope of the present manual. You should refer to the *ZeBu Compilation Manual* and the *ZeBu Tutorial Manual* for information about DUT compilation with **`zCui`**.

To compile the ZEMI-3 transactor with **`zCui`**, right-click on **Sources** in the **Project Tree** pane, choose **Add Zemi-3 Transactor...** and type the name of your ZEMI-3 transactor (default name is `my_zemi3_xtor`). This operation creates an item in the **Project** → **Sources** tree, as for the design. When creating the transactor, **`zCui`** automatically creates the appropriate items to add the source files in an RTL group (`ZEMI3_RTL_group` created by default) or as EDIF source files. You can also add some memory Tcl description for **`zMem`**.

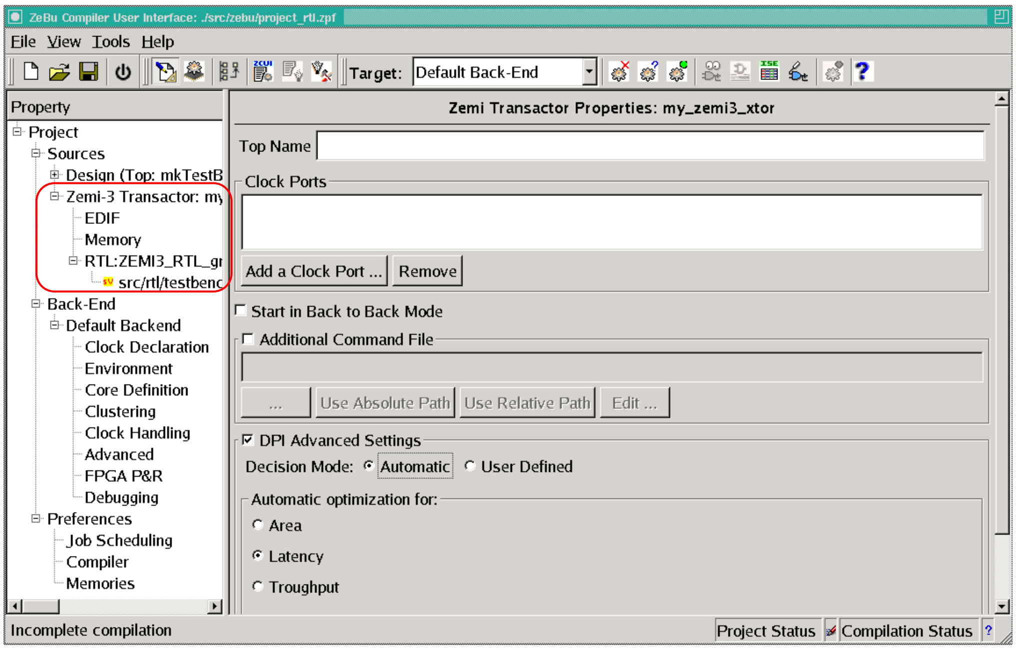


Figure 14: `zCui` screenshot for a ZEMI-3 transactor

In the **ZEMI-3 Transactor Properties** page, provide the following information:

1. In the **Top Name** field, you have to declare the top name of the transactor.
2. In the **Clock Ports** area, you add/remove the names of the controlled clocks which are connected to the transactor, as they are declared in the DVE file.
3. Check the **Start in Back to Back Mode** box if you want to start your emulation in *back-to-back mode* for easier debug of your transactor (see Section 2.4.1).
4. Check the **Additional Command File** box if you want to declare an attribute for **`zEmiComp`**. For example:

```
Compile:Zxtor:AsyncStopClock=false
```

5. In the **DPI Advanced Settings** area, specify a **Decicion Mode**:
   a. If you choose **Automatic**, select **Automatic optimization for** any of the following options (only one option can be selected at a time):
      - **Area**, to minimize the usage of FPGA resources
      - **Latency**, to minimize the HW-SW latency
      - **Throughput**, to maximize the data throughput

   b. If you choose **User Defined**, specify the DPI advanced settings for message multiplexing and/or message buffering.

4.4 Results of the ZEMI-3 transactor pre-processing

The `zEmiComp` log file is displayed in the `zCui` right panel. It shows information for each of the inter-language functions.

For each import/export function, the elements listed in the logfile are the following:

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`)

The list of message ports at the bottom of the logfile is intended for advanced optimization of your transactional environment.

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)
INFO:
```

4.5 Compilation output files for ZEMI-3 runtime

For each transactor, `zCui` compilation generates a dynamic library (`<my_xtor_dve>.so`) which is stored in the `zebu.work` directory, with the corresponding header file (`<my_xtor_dve>.h`). `<my_xtor_dve>` is the module name of the ZEMI-3 transactor in the DVE file.

5 Running ZEMI-3 Environment

5.1 Introduction

The ZEMI-3 environment is very similar to a pure simulation environment: a generic executable called **`zEmiRun`** handles the emulation automatically. User has to provide the implementation of the import calls and optionally a testbench to control the emulation. A testbench is especially useful for export tasks which consume simulation time.

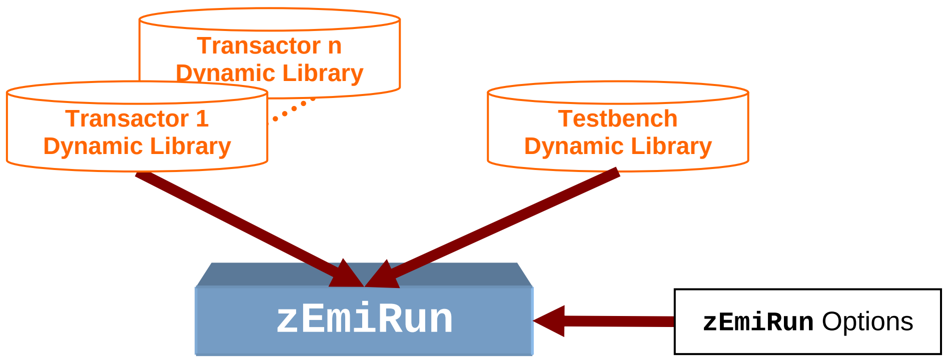


Figure 15: ZEMI-3 runtime environment

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

- Using an existing SystemC SW testbench environment (see Section 6.1)
- Mixing zcei transactors and ZEMI-3 transactors (see Section 6.2)

5.2 `zEmiRun` requirements

- Since **`zEmiRun`** is a thread-based application, it is highly recommended to run it on a multi-processor host to get the best performance.
- If the compilation is performed on a PC configuration different from the emulation runtime, the dynamic libraries can be recompiled with the `dpixtor.mk` makefile in the `zebu.work` directory. This makefile has two different targets:
  - `all`: compile all the dynamic libraries and copy them in `zebu.work`
  - `clean`: clean the dynamic libraries
- The following can be used to override the makefiles generated by **`zEmiComp`**:
  - `LDFLAGS` to add linker options [default: `empty`]
  - `CXXFLAGS` to add compiler options [default: `-O`]
  - `ZEMI3_LIB` to change the zemi3 lib [default: `ZebuZEMI3`]
  - `XTOR_LIB` to rename the generated *`.so` file
    [default: `<xtor_mod_name>.so`]

5.3 User-provided files for runtime

5.3.1 Dynamic library

User has to provide:

- The implementation of the import calls
- The implementation of the testbench function

The import and export functions as well as the testbench functions have to 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 file generated by the **`zCui`** compilation (`<my_xtor_dve>.h`).

The prototype of the import and export functions can be found in each transactor header file, `<my_xtor_dve>.h`, in the `zebu.work` directory. The name of the header file matches the ZEMI-3 transactor module name in the DVE file.

5.3.1.1 Testbench function

User can optionally provide a C function performing the calls to export tasks in order to control emulation. This so-called "testbench function" is called by the `ZEMI3Manager` object. When this function returns, the emulation terminates.

This testbench function has to be declared as `extern "C"`. It can call export tasks that consume simulation time.

The testbench function prototype is the following:

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

Where returned `int` is the returned status of **`zEmiRun`**.

If no testbench function is provided, then unless some other termination mechanism (e.g. fixed time, number of clock cycles) is specified, the emulation will not terminate.

5.3.1.2 Compilation of the user dynamic library

The user dynamic library has to be linked with the `libZebuZEMI3.so` library.

The `–rdynamic` and `–fPIC` options should be used when the test contains import functions.

The example below shows how to compile the user 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
```

5.4 Launching zEmiRun

5.4.1 Standard usage

The name of the testbench function and the library to which it belongs can be specified to **zEmiRun** with the -m option: -m <user_dyn_lib>[:<tb_func>]

Example:
```
$ zEmiRun -m user_testbench.so:my_testbench_function
```

zEmiRun can also be launched without a testbench function.

Example:
```
$ zEmiRun -m user_testbench.so
```

5.4.2 zEmiRun options

Table 4: zEmiRun options

Note: at least one dynamic library must be specified with option -l, -m, -i or –b.

zEmiRun option	Description
-z <zebu.work>	Specifies the directory for the design database. Default is ./zebu.work
-f <designFeatures>	Specifies the file containing the runtime configuration. This file contains information regarding clocks and clock groups. No default
-p <processName>	Specify a name for the process. This is used if a multi-process emulation is being run. No default
-v	Sets the verbose mode. This turns on the printing of some potentially interesting messages during the run. Default is no verbose
-c <xtor_configuration>	Specifies a configuration file for the ZEMI-3 transactors. The default is the zCui generated file ./zebu.work/xtor_dpi.lst. This file lists all the transactors to load with each transactor listed in a single line in the following format: <xtorModuleName> <xtorInstanceName> [<xtorDynamicLib>] The dynamic library (or object) can be left unspecified if it is later specified by the -x option. The latter will also override any specification here.
-x <xtor_dyn_lib>:<xtor_module>	Specifies a dynamic library (object) to be used for the transactor implementation. This option is not normally used. It can be used if the files are moved out of the standard location and the files specified in the transactor configuration point to the wrong (old) place.
-l <usr_dyn_lib>	This option is used to load a user compiled dynamic library in the global scope. This library may contain implementations of the import functions used in the transactors and/or the

`zEmiRun` option	Description
	`main/init/callback` functions specified on the command line. Multiple `-l` options can be specified. If the library refers to any symbol from the transactor objects (e.g. export functions), then care should be taken to compile/link it with the `-fPIC` compiler option, otherwise it will fail to load. Also note that the dynamic libraries, if any, specified with the `-m`, `-i` or `-b` options are also loaded in the global scope, so this option is not necessary if all libraries needed are specified with one of the other options.
`-m <usr_dyn_lib>[:<tb_func>]`	Specifies a function to be called as the user main function. If a function is not specified, then this option behaves the same as the `–l` option. Only one `-m` option can be specified. The function also gets passed the command line arguments specified after the `'--'` flag. The return value of this function is used as the exit status from **`zEmiRun`**. If an entry point is specified, then the function is resolved only within this file. The library is also loaded globally and is available to resolve other global symbols.
`-i [<usr_dyn_lib>:]<init_func>`	Specifies a transactor initialization code (e.g. set BackToBack mode, set prefetch mode, etc.) to call just after the opening of the board and before the import processing starts, and an optional dynamic library where to find it. Calling export functions is not prohibited here, but it is possible to get into a deadlock situation if the message ports are not being serviced. Multiple `-i` options can be specified. All the functions also get passed the command line arguments specified after the `'--'` flag. If any function returns a non-zero value, **`zEmiRun`** treats it as a signal to abort the run. If the dynamic library is specified, then the entry function will only be resolved within that library. Thus it can be used to choose a specific function. The library is also loaded globally and is available to resolve other global symbols. However if the function is to be resolved globally, the dynamic library may be left unspecified if it is already specified with `-m`, `-b` or `-l`. See also the note under the `-l` option about using the `-fPIC` compiler option.
`-b [<usr_dyn_lib>:]<callback_func>`	Specifies a function to be called back periodically during the emulation. Although the import processing is active at the time this function is called, deadlock is still possible if export functions are called from the callback function.

`zEmiRun` option	Description
	This is a useful entry point for scoreboard or monitoring operations. Although there are no guarantees about when the callbacks will be called, the first call will always be made right before the import processing has begun. Multiple `-b` options can be specified. These functions are not passed any arguments. The return value is interpreted as follows: • Zero (`0`): disable this callback • Non-Zero (`N`): reschedule this callback after `N` iterations of the service loop If the dynamic library is specified, then the entry function will only be resolved within that library. Thus it can be used to choose a specific function. The library is also loaded globally and is available to resolve other global symbols. However if the function is to be resolved globally, the dynamic library may be left unspecified if it is already specified with `-m`, `-b` or `-l`. See also the note under the -l option about using the `–fPIC` compiler option.
`-n`	No use of threads. Only available if no user main
`-r <clock>:<cycle_number>`	Specifies the number of cycles to perform on clock `<clock>` before exit if no user main function
`-t <time in seconds>`	Specifies a time before exit if no user main function
`-- <user main arguments>`	Specifies arguments passed to user main function

5.4.3 Launching **zRun** with **zEmiRun**

In order to use the usual ZeBu debugging features, in particular waveform dumping, user can launch **zRun** with **zEmiRun**:

```
$ zRun –testbench "zEmirun –z ../zebu.work
                   –m user_testbench.so:my_testbench_function"
```

5.4.4 Activating the profiling feature

By default, profiling is turned ON. To turn it OFF, use the following attribute via the **Additional Command File** option in **zCui**:

```
Compile:Zxtor:DoProfiling=false
```

6 Running in a Heterogeneous Environment

Two advanced runtime environments can also be used with a ZEMI-3 transactor:

- Using an existing SystemC SW testbench environment (see Section 6.1).
  This mode gives access to a `ZEMI3Manager` object for emulation control.
- Mixing zcei transactors and ZEMI-3 transactors (see Section 6.2).

6.1 Using a SystemC environment

In a SystemC environment, the runtime is handled by the SystemC kernel. Since **`zEmiRun`** contains a scheduler similar to SystemC, they cannot be used together. In a SystemC environment, the ZEMI-3 environment has to be initialized by user.

In such an environment, the user testbench uses a C++ class which initializes the transactors.

6.1.1 `ZEMI3Manager` class

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

The following sections describe the procedure for environment initialization.

6.1.1.1 `ZEMI3Manager` creation

The constructor of the `ZEMI3Manager` object also performs the open of the ZeBu Board.

```
ZEMI3Manager(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`: process name

Example:

```
ZEMI3Manager *dm = new ZEMI3Manager("./zebu.work", ". /designFeatures",
"processName");
```

6.1.1.2 Transactor list specification

The list file is automatically generated in the `zebu.work` directory by **`zFW`** as `xtor_dpi.lst`. The name of this file has to be declared for the runtime environment:

```
void buildXtorList(const char *listFile);
```

Example:

```
dm->buildXtorList(my_xtorFile);
```

6.1.1.3 Initializing the `ZEMI3Manager` object

This is the last step in the initialization process of the `ZEMI3Manager`: it starts the clocks and launches a thread for each transactor. The launched threads are calling the transactor imports.

```
dm->init();
```

6.1.1.4 `ZEMI3Manager` advanced features

6.1.1.4.1 Get the `ZEBU::Board` object

It is possible to get the `ZEBU::Board` object from the `ZEMI3Manager` object. This is useful for runtime advanced features such as logic analyzer or clock control.

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

Example:

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

6.1.1.4.2 Back-to-back control

As in stand-alone mode, it is possible to control the *back-to-back mode* from the `ZEMI3Manager` object and the functions are the same as described in Section 3.2.5.

6.1.1.5 Error handling

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

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);
}
```

6.1.2 Compilation

The final executable has to be linked with `libZebuZEMI3.so`.

```
$ g++ -fPIC -c dpi_ctrl.cc -Izxtor/my_xtor_1 -Izxtor/my_xtor_2 \
  -I$ZEBU_ROOT/include
$ g++ -o user_tb *.o -L$ZEBU_ROOT/lib zxtor/my_xtor_1/my_xtor_1.so\
  zxtor/my_xtor_2/my_xtor_2.so –lZebuZEMI3
```

6.2 Using zcei transactors and ZEMI-3 transactors

6.2.1 Introduction

It is possible to mix ZEMI-3 transactors and zcei transactors. In this case the user will have a zcei-based environment or a ZEMI3-based environment. As in standard zcei-based environments all ZeBu initializations have to be done via `ZEBU::Board` class.

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

The ZEMI-3 transactor base class is very simple in order to:

- Connect/disconnect a ZEMI-3 transactor
- Serve the imports
- Start/stop the *back-to-back mode*

6.2.2 `ZEMI3Xtor` class

```
#ifndef _ZEMI3Xtor_hh_
#define _ZEMI3Xtor_hh_

#ifdef __cplusplus

#include "XtorBase.hh"

namespace ZEBU
{

class Board;
class ImportHandler;
class ExportHandler;
class ZEMI3XtorImp;

class ZEMI3Xtor : public ZEMI3XtorBase
{
protected:
    // The following will be called from constructor of final derived class
    void addImport (ImportHandler *handler, const char *rxPortName, unsigned int rxWidth,
            const char *txPortName, unsigned int txWidth, const char *relPath);
    void addExport (ExportHandler *handler, const char *rxPortName, unsigned int rxWidth,
            const char *txPortName, unsigned int txWidth, const char *relPath);
    void addPort (const char *portType, const char *uniqueName,
            unsigned int id, const char *portName, unsigned int width,
            const char *sharingName, const char *relPathPrefix);
    void setControlStatusInfo (unsigned int start, const char *controlPortName, const char
*statusPortName) ;

public:
    ZEMI3Xtor (const char *xtorName) ;
    virtual ~ZEMI3Xtor () ;

    // Initialize the transactor, Connect Zebu ports
    // Should be done before Board::init()
    /*redef*/ void init (Board *zebu, const char *instancePath) ;

    // Close the transactor, Disconnect Zebu ports
    // Should be done before Board::close()
    /*redef*/ void close () ;

    // Check whether there are any Zebu ports
    // that need service via a service loop
    /*redef*/ bool needsService () const;

    // Register import handlers so that they can be serviced
    // via the Board::serviceLoop()
    void setGroup(unsigned int portGroup);

    /*redef*/ void registerImports(unsigned int portGroup=1);
    /*redef*/ void unregisterImports();

    // Check whether there is any message pending for any
    // of the import handlers
    /*redef*/ bool checkImports () ;

    // Service the Zebu ports. Dispatch messages by calling the
    // import functions
    /*redef*/ void serviceLoop () ;
    void serviceLoopWithWait (int timeout=0) ;

    // re-enable the processing of imports/exports. This is useful
    // after a terminate, if you want to restart processing
    /*redef*/ void reset () ;

    // terminate any ongoing processing. This will break
    // any ongoing service loop
    /*redef*/ void terminate () ;

    // Obtain some compile time information regarding
    // the transactor instance
    /*redef*/ xtorInfos* getXtorInfos() ;

    // transactor control/status operations
    /*redef*/ void startBackToBack () ;
    /*redef*/ void stopBackToBack () ;
    /*redef*/ bool isBackToBackStarted () ;

    /*redef*/ void disableImports () ;
    /*redef*/ void enableImports () ;
    /*redef*/ bool isImportDisabled () ;

    /*redef*/ void flushBufferedPorts ();

    /*redef*/ bool getErrorStatus ();
    /*redef*/ bool getExportActiveStatus ();

protected:
    ZEMI3XtorImp *_xtor;
};

} // of namespace

#endif // __cplusplus

/* C-Interface */

#ifdef __cplusplus
extern "C" {
#endif

// Control Back2Back function of a transactor instance
void ZEMI3_startBack2Back(const char *xtorName);
void ZEMI3_stopBack2Back(const char *xtorName);
int ZEMI3_isBack2BackStarted(const char *xtorName);

// Other control operations
void ZEMI3_disableImports (const char *xtorName);
void ZEMI3_enableImports (const char *xtorName);
int ZEMI3_isImportDisabled (const char *xtorName);
void ZEMI3_flushBufferedPorts (const char *xtorName);
int ZEMI3_getErrorStatus (const char *xtorName);
int ZEMI3_getExportActiveStatus (const char *xtorName);

#ifdef __cplusplus
}
#endif

#endif
```

6.2.3 Using a ZEMI-3 transactor in a zcei testbench

The `<my_xtor_dve>.h` header file, generated by **`zCui`** in the `zebu.work` directory, has to be included in the user testbench.

This file declares the user transactor interface class, which inherits from the base class of the transactor `ZEMI3Xtor` class. To handle the ZEMI-3 transactor, an object of this transactor class has to be instantiated in the testbench.

6.2.3.1 Creation of the transactor object

An object has to be created for each instance of the transactor instantiated in the DVE file. The object has to be created just between the `ZEBU::Board::open` and `ZEBU::Board::init`.

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

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

6.2.3.2 Initializing a ZEMI-3 transactor

The `init()` method initializes the transactor by connecting its ports. It has to be done between the creation of the transactor object and `ZEBU::Board::init`. The parameters are the `Board` object and the transactor instance name (in the DVE file).

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

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

6.2.4 Running a ZEMI-3 transactor

When interfacing a ZEMI-3 transactor with a zcei environment, the export functions are called explicitly by user in the testbench and import functions are serviced by the transactor service loop method. In our case, there are two ways to call the imports:

- Using the transactor local `ZEMI3XtorBase::serviceLoop()` call. This will directly check if there are messages available on the message port(s).
- The second is the callback method. In this mode, a callback is registered for the import and is triggered when `Board::serviceLoop()`is called.

Warning: Up to release 4.3.2 with patch B_00, both mechanisms were ON by default. This led to the case where if the user wanted to use the transactor service loop for the ZEMI-3 transactor, but wanted to use the board service loop for other processing, the transactor import callbacks were being triggered and the user could not stop them.

Thus, the default was changed to NOT register callbacks for the imports. In addition, the `ZEMI3XtorBase::registerImports()` method has been added to allow the user to do the callback registration if desired.

This puts the user in control of how he wants the transactor imports to be handled. It is still possible though, after registering the imports, to have both the board service loop and the transactor service loop running. It is however not advisable to do so as it may impact performance due to extra thread safety (mutex) issues.

6.2.4.1 Using the local transactor `serviceLoop`

When called, `ZEMI3XtorBase::serviceLoop` checks if an import has been called from the HW 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 thread serving only the transactor import. When using an application with one thread per transactor, `ZEMI3XtorBase::serviceLoop` should be used.

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(ZEMI3XtorBase *xtor)
{
  while(!TEST_END) xtor->serviceLoop();
}

int main()
{
  Board *z = Board::open();

  // Initialization of the ZEMI-3 transactors
  ZEMI3xtor *zemi3_xtor_0 = new ZEMI3_xtor();
  ZEMI3_xtor *zemi3_xtor_1 = new ZEMI3_xtor();
  zemi3_xtor_0->init(z, "zemi3_xtor_0");
  zemi3_xtor_1->init(z, "zemi3_xtor_1");

  // Initialization of the zcei transactor
  ZCEI_xtor *zcei_xtor_0 = new ZCEI_xtor(z, "zcei_xtor_0");

  // ZeBu init
  z->init();

  // Launching threads for the ZEMI-3 transactors
  pthread_t tid1;
  pthread_create(&tid1, NULL, (void *(*)(void *))xtorImportServer, (void
*)dpi_xtor_0);

  pthread_t tid2;
  pthread_create(&tid2, NULL, (void *(*)(void *))xtorImportServer, (void
*)dpi_xtor_1);

  // testbench execution
  ...
  // disconnection
  TEST_END = true;
  zemi3_xtor_0->disconnect();
  zemi3_xtor_1->disconnect();
  z->close();
}
```

6.2.4.2 Using the `ZEBU::Board` `serviceLoop`

User will need to register the imports of each transactor that he wishes to be handled by the `Board::serviceLoop`. This provides some flexibility. See below for 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 of the ZEMI-3 transactors
  ZEMI3_xtor *zemi3_xtor_0 = new ZEMI3_xtor();
  ZEMI3_xtor *zemi3_xtor_1 = new ZEMI3_xtor();

  zemi3_xtor_0->init(z, "zemi3_xtor_0");
  zemi3_xtor_0->registerImports ((int)zemi3_xtor_0);
  zemi3_xtor_1->init(z, "zemi3_xtor_1");
  zemi3_xtor_1->registerImports ((int)zemi3_xtor_1);

  // Initialization of the zcei transactor
  ZCEI_xtor *zcei_xtor_0 = new ZCEI_xtor(z, "zcei_xtor_0");

  // ZeBu init
  z->init();

  // Launching DPI xtor threads
  pthread_t tid;
  pthread_create(&tid, NULL, (void *(*)(void *))xtorImportServer, (void
*)z);

  // testbench execution
  ...

  // disconnection
  TEST_END = true;
  zemi3_xtor_0->disconnect();
  zemi3_xtor_1->disconnect();
z->close();
}
```

6.3 Performing simulation of ZEMI-3 transactors

A ZEMI-3 transactor and his stand-alone environment can be simulated with any SystemVerilog compliant simulator.

6.3.1 Writing a top-level wrapper for simulation

In emulation, the connection between the clock ports, the DUT and the transactor is performed through the DVE file.

In pure simulation, it is necessary to write a wrapper which describes these connections and generates the clocks.

Example:

```
module 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)
                     );
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, top);
    $dumpon;
  end
endmodule
```

6.3.2 Using a SW entry point in pure simulation

The only difference with emulation comes from the testbench function which is not available in simulation.

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

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

6.3.3 Launching the simulation

This section describes the command lines for standard HDL simulators. The filenames match the memory transactor example described in Section 6.3.1.

6.3.3.1 Using ModelSim

Compilation for the HDL simulator:

```
$  vlib work
$  vlog  -dpiheader mem_xtor.h  ../src/*.sv ../src/*.v \
   +define+SIMULATION
```

gcc compilation of user C code (`-g` for debug symbols):

```
$  g++ -c -g -fPIC -I$(MTI_HOME)/include ../src/tb.cc -I../src -I.
$  g++ -c -g -fPIC -I$(MTI_HOME)/include ../src/memTester.cc \
   -I../src/ -I.
$  g++ -shared -o tb.so *.o
```

To run the simulation:

```
$ vsim  top -sv_lib tb -c -do "run -all"
```

6.3.3.2 Using VCS/VCSi

You need to create a `mem_xtor.h` header file which includes the prototypes of the export functions:

```
#include <svdpi.h>
extern "C" export (...);
#define XTOR_SCOPE "top.xtor_0"
```

Compilation for the HDL simulator:

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

To run the simulation:

```
$ ./simv
```

6.3.3.3 Using NC-Verilog

You need to create a `mem_xtor.h` header file which includes the prototypes of the export functions:

```
#include <svdpi.h>
extern "C" export (...);
#define XTOR_SCOPE "top.xtor_0"
```

gcc compilation of user C code (`-g` for debug symbols):

```
$ g++ -c -g -fPIC -I$(NC_HOME)/include ../src/tb.cc -I../src -I.
$ g++ -shared -o libdpi.so *.o
```

Compilation for the HDL simulator:

```
$ ncvlog -sv ../src/*.sv ../src/*.v
$ ncelab top
```

To run the simulation:

```
$ ncsim top
```

7 Tutorial

This chapter presents several examples of ZEMI-3 based transactors. The source code examples for the HW and SW parts of the transactor have detailed explanations. For each example, a functional description is provided, and the specific ZEMI-3 features which correspond are highlighted.

The complete source code files and scripts for these examples are available in a separate package.

7.1 Tutorial architecture

Typical tutorial architecture:

```
zemi3_example_X
|-- Makefile
|-- README
`-- src
    |-- dut.dve
    |-- dut.v
    |-- tb.cc
    |-- xtor.sv
    `-- zcui.zpf
```

The makefile targets are the following:

```
make compil_batch : launch zCui in batch mode
make compil_gui : launch zCui in gui mode
make run : launch runtime
make simul_vcs : launch simul_vcs
make simul_mti : launch simul_mti
make simul_nc : launch simul_nc
make clean : clean all
```

7.2 Initializing input configuration registers

7.2.1 Functional description and ZEMI-3 highlighted features

In this example, the SW controls the initialization of registers which are mapped in the HW part of the transactor. These registers control writing operations into the DUT, which is a memory module.

With the ZEMI-3 infrastructure, the initialization of the registers is initiated by the SW, using an export function. There is no need for clock cycles to perform the operations on the HW side, therefore it is not necessary to use an export task.

The inputs of the export function are the values to write in the configuration registers. Since there are no outputs or returned values, ZEMI-3 automatically sets this export function as a streaming function. The execution of the C function returns before the HW part requires the data.

This example demonstrates how to use export functions and how the *streaming mode* improves the performance of the HW part of the transactor when the SW part of the transactor sends data to the HW.

7.2.2 Implementation

7.2.2.1 Architecture

The test consists in a transactor writing values into a memory. Three parameters need to be set in the transactor:

- First address to write
- Number of addresses to write
- Initial value of the counter written into memory

The initialization of the registers is done by an export function taking as argument the address, the number of addresses and the initial value of the counter. The export function also sets a start register to 1 to activate the transactor. When all the specified addresses have been written, the start register is set to 0.

The test will be launched in stand-alone mode with **`zEmiRun`**. The testbench function calling the export will be called `zemi3_main`.

In stand-alone mode, **`zEmiRun`** automatically calls a user-defined testbench function which has to be specified in the **`zEmiRun`** command line. This user-defined testbench function has to be declared as `extern "C"`.

7.2.2.2 DUT description

The DUT memory has the following interface:

```
module dut( input        clk,
            input        we,
            input  [63:0] data_in,
            input  [19:0] addr,
            output [63:0] data_out);
```

Where:

`clk` is the clock of the memory, implicitly controlled by the transactor
`we` is the write enable high of the memory
`data_in` is the input data of the memory
`addr` is the address where the data are written
`data_out` is the output data of the memory

7.2.2.3 HW part of the transactor

The transactor interface will be the following:

```
module xtor(  input               clk,
              output reg          we,
              output reg [19:0]   addr,
              output reg [63:0]   data);
```

`clk` is the clock of the transactor. This clock will be implicitly controlled by the transactor. It needs to be declared in **`zCui`** in the **Clock Ports** list of the **ZEMI-3 Transactor Properties** tab, as described in Section 4.3. The other outputs are connected to the ports of the DUT.

The following registers and variables are declared in the transactor module:

```
reg start;
integer nb;
reg [19:0] start_addr;
reg [63:0] start_data;
reg [19:0] nb_addr;
```

The `start` register will be set by the export function to activate the writing operation. The 3 registers `start_addr`, `start_data` and `nb_addr` are the configuration registers set by the export function.

The declaration of the ZEMI-3 export function is the following:

```
export "DPI-C" function initialize_registers;
```

The arguments are defined as bits. It means that the values taken by the arguments can only be 0 or 1. With bit arguments, the C type of the parameters will be `svBitVecVal`. If the arguments were untyped, the C interface of the function would be a `svLogicVecVal` which is a 4-state type. The arguments types are defined in the `svdpi.h` file.

```
function void initialize_registers(input bit [19:0] nb_addr_arg,
                                   input bit [19:0] start_addr_arg,
                                   input bit [63:0] start_data_arg);
  begin
    start_addr  = start_addr_arg;
    start_data  = start_data_arg;
    nb_addr     = nb_addr_arg;
    start       = 1;
  end
endfunction;
```

The export function sets the configuration registers of the transactor and has no return value or outputs.

The registers of the transactor are initialized in an `initial` block:

```
initial begin
    we         <= 0;
    addr       <= 0;
    data       <= 0;
    start       = 0;
    start_addr  = 0;
    start_data  = 0;
    nb_addr     = 0;
end
```

When the `start` register is set to 1 by the export function, the writing operation to the memory will start. This process will write from address `start_addr` to address `start_addr+nb_addr` the values of a counter incremented at each posedge of `clk`.

This counter is initialized by the export function.

```
  always @(posedge start) begin
    addr <= start_addr;
    data <= start_data;
    @(posedge clk);
    for(nb=0;nb<nb_addr;nb=nb+1) begin
      we <= 1;
      addr <= start_addr + nb;
      data <= data + 1;
      @(negedge clk);
      @(posedge clk);
    end
    we <= 0;
    start = 0;
  end
endmodule
```

7.2.2.4 DVE file

The DVE file corresponding to this Lab is the following:

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

xtor xtor_0(.clk(clk),
            .addr(addr[19:0]),
            .data(data_in[63:0]),
            .we(we));
```

Note: Unlike in Verilog, the syntax of the DVE file requires that the DUT ports ranges be explicitly declared in case of vectors.

7.2.2.5 SW part of the transactor

The SW part of this test will be launched by **`zEmiRun`**. This tool needs a dynamic library containing the import functions implementation and the testbench function. In this example the testbench function calls the export function `initialize_registers`.

The C file which includes the testbench function is `tb.cc`.

The following header files are included:

- The header file generated by **`zCui`** for the transactor (`xtor.h`): it contains the declaration of the imports and exports prototypes. Including this file is strongly recommended because the C compiler will check the compliance of the calls between the user code and the declared prototypes. This file is generated in the `zcui.work/zebu.work` directory.
- The header file for the standard SystemVerilog API (`svdpi.h`): it defines the standard C types and utility functions for the DPI.

The testbench function takes no argument and returns an integer (`int`) which will be the return value of **`zEmiRun`**. This function needs to be compiled with `gcc` or specified as `extern "C"` if compiled with `g++` (if not, you won't be able to run your environment with **`zEmiRun`**).

```
extern "C" int zemi3_main()
{
```

The declaration of the variables of the export function is the following:

```
  /* Declaration of variables which initialize HW */
  svBitVecVal start_addr[SV_PACKED_DATA_NELEMS(20)];
  svBitVecVal start_data[SV_PACKED_DATA_NELEMS(64)];
  svBitVecVal nb_addr[SV_PACKED_DATA_NELEMS(20)];
```

The type of the parameters of the export function is `svBitVecVal` because the inputs of the export have been declared as bits.

Before calling an export task the current scope has to be set to the appropriate value, which is the instance name of the transactor in the DVE File ("xtor0" in this example):

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

Initializing the variables to be sent to the HW:

```
  start_addr[0] = 789;
  nb_addr[0]    = 100;
  start_data[0] = 0xBABE0000;
  start_data[1] = 0xFACE1234;
```

Calling the export function:

```
initialize_registers(nb_addr, start_addr, start_data);
```

Note: In the current example, the testbench contains only a call to the export function. When the testbench function returns, the emulation stops. This means that the SW part sometimes returns before the HW part has completed the run.

7.2.3 Compiling and running

The `tb.cc` file has to be compiled in a dynamic library (`tb.so`) for **`zEmiRun`**:

```
$ g++ -fPIC -shared -o tb.so ../src/tb.cc –I../
  zcui.work/zebu.work/ -I$ZEBU_ROOT/include/ -L$ZEBU_ROOT/lib
  -lZebuZEMI3
```

The following command launches the test:

```
$ zEmiRun -z ../ zcui.work/zebu.work -m ./tb.so:zemi3_main
```

7.2.4 Advanced runtime usage

7.2.4.1 Getting waveforms for debugging with **`zRun`**

`zEmiRun` can be used with **`zRun`** to dump waveforms. The following command launches **`zRun`** and **`zEmiRun`**:

```
$ zRun –testbench "zEmiRun -z ../ zcui.work/zebu.work
  -m ./tb.so:zemi3_main"
```

7.2.4.2 Pure simulation of the testcase

This test can be easily simulated with an HDL simulator. The DVE file which connects the clocks and the DUT in the ZeBu environment has to be replaced by a dedicated wrapper for pure simulation.

This file contains the generation of the clock, the call of the testbench function and the instantiations of the DUT and of the transactor.

```
module top();

reg clk;
wire [19:0] addr;
wire [63:0] data;
wire we;

import "DPI-C" context task testbench();
```

```
initial #100 testbench();

initial clk <= 0;

initial begin
  $dumpfile("simu.vcd");
  $dumpvars(0, top);
  $dumpon;
end

always #10  clk <= ~clk;

dut dut_0(.clk(clk),
          .addr(addr),
          .data_in(data),
          .we(we),
          .data_out()
         );

xtor xtor_0(.clk(clk),
            .addr(addr),
            .data(data),
            .we(we)
            );
endmodule
```

7.3 Polling the HW status

7.3.1 Functional description and ZEMI-3 highlighted features

This example demonstrates how to use an export function so that the SW can check the status of the HW part of the transactor.

Since the SW initiates the transaction, the polling operation uses an export. A function can be used because reading the value of a register does not consume simulation time. The export function has only outputs.

In ZEMI-3, an export with only outputs generates two messages:

- SW-to-HW message to initiate the call of the SystemVerilog function
- HW-to-SW message to return the output data

The example is in stand-alone mode.

7.3.2 Implementation

7.3.2.1 Architecture

The transactor implemented in this example can have two states:

- `IDLE`: the clocks are running with no other activity
- `UNDER_TEST`: the transactor is sending data to the DUT

By default the transactor is in `IDLE` state. When the function `start_test` is called, the transactor goes into `UNDER_TEST` state. In `UNDER_TEST` state, the transactor is sending value of the counter to the DUT. When the counter reaches the number of cycles set by the `start_test` function, the transactor goes back into `IDLE` mode.

The testbench function calls the `start_test` export function and checks periodically the status of the transactor with the export task `getXtorStatus`. When the testbench detects that the transactor is back in `IDLE` mode, the test terminates.

Two exports are necessary for that:

- `start_test`: initiates the `UNDER_TEST` sequence using an event and sets the number of cycles for the `UNDER_TEST` sequence
- `getXtorStatus`: reads the state of the transactor and the clock counter

7.3.2.2 HW part of transactor

The two states of the transactor are defined at the beginning of the SystemVerilog source code of the transactor:

```
`define IDLE         1'b0
`define UNDER_TEST   1'b1
```

The interface of the transactor is the following:

```
module transactor(input clk, output reg [31:0] data_out);
```

`clk` is the clock of the transactor. This clock will be implicitly controlled by the transactor. It needs to be declared in **`zCui`** in the **Clock Ports** list of the **ZEMI-3 Transactor Properties** tab, as described in Section 4.3. The other outputs are connected to the ports of the DUT.

The following registers and variables are declared in the transactor module.

```
  integer i;
  event start;
  reg        xtor_state;
  reg [31:0] clock_counter;
  reg [31:0] cycle_max;
```

Where:

- `start` is the Verilog event which activates the `UNDER_TEST` sequence
- `clock_counter` counts the number of clock cycles during the `UNDER_TEST` sequence
- `xtor_state` is the state register of the transactor.
- `cycle_max` is the number of cycles to perform during the `UNDER_TEST` sequence

The task `start_process` asserts an event which starts sending data to the DUT.

This task takes as argument the number of cycles to perform during the `UNDER_TEST` sequence.

The declaration and implementation of the export task is the following:

```
  export "DPI-C" task start_test;
  task start_test(input bit [31:0] cycle_number);
  begin
    cycle_max = cycle_number;
    -> start;
  end
  endtask
```

The export function returning the HW state to the SW is described below. The first parameter is the state of the transactor and the second is the number of clock cycles performed by `clk` since the beginning of the emulation.

Note: Blocking assignments are used for output assignments, as in a traditional function.

```
export "DPI-C" function getXtorStatus;

function getXtorStatus (output bit s, output bit [31:0] counter);
  begin
    s = xtor_state;
    counter = clock_counter;
  end
endfunction
```

Initially the `data_out` register is set to 0 and the state to `IDLE`.

```
initial begin
    data_out      <= 0;
    xtor_state    <= `IDLE;
    clock_counter <= 0;
    cycle_max      = 0;
end
```

When the `start` event is asserted, the process sets the transactor state register to `UNDER_TEST` and increments the `data_out` register connected to the DUT. When the number of cycles to execute is reached, the transactor comes back in `IDLE` mode.

```
  // ------------- Stimulation of the DUT
  always
  begin
    @(start);
    local_counter <= 0;
    xtor_state <= `UNDER_TEST;
    while (local_counter < 32'h00008000) begin
      data_out <= data_out + 1;
      @(posedge clk);
    end
    xtor_state <= `IDLE;
    data_out <= 0;
  end
```

7.3.2.3 DVE file

The DVE file corresponding to this Lab is the following:

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

transactor xtor0 (
     .clk ( clk ),
     .data_out ( data_out[31:0] )
);
```

Note that, unlike Verilog, the syntax of the DVE file requires that the ranges of the DUT ports are explicitly declared in case of vectors.

7.3.2.4 SW part of the transactor

The testbench function initiates the process with the `start_test` task and checks the status of the HW from time to time. When the HW part comes back in `IDLE` mode, the testbench function returns.

The C file which includes the testbench function is `tb.cc`.

The header file generated for this transactor has to be included in the C source file:

```
#include <transactor.h>
#include <stdio.h>
```

Defines having the same value as the HW defines are declared.

```
#define IDLE       0
#define UNDER_TEST 1
```

The testbench function is declared as extern “C”.

```
extern "C" int zemi3_main()
{
```

The declaration of the parameters of the export functions is the following:

```
  svBit status;
  svBitVecVal counter[1];
  svBitVecVal nb_cycles[1];
```

The scope is set according to the instance name of the transactor in the DVE file:

```
  svScope s = svGetScopeFromName("xtor0");
  svSetScope(s);
```

The command that starts the transactor in `UNDER_TEST` mode is called. The number of controlled clock cycles to perform before reaching the `IDLE` state is set to 100,000.

```
  nb_cycles[0] = 100000;
  start_test(nb_cycles);
```

In the following sequence, the state register is polled until it reaches `IDLE`.

```
  printf("# STARTING TEST \n");
  do {
    /* Any operation can be performed here */
    /* For this example usleep stands for any operation */
    usleep(10);
    getXtorStatus(&status, counter);
    printf("# At cycle %d status = %s\n", counter[0],
status==IDLE?"IDLE":"UNDER_TEST");
```

```
  }
  while (status == UNDER_TEST);
  printf("# TEST COMPLETED\n");
  return 0;
  }
```

Notes:

- The operation performed between two calls is a `usleep` but should be replaced in a real life testcase by any useful operation
- Removing the `usleep` and calling the `getXtorStatus` function at each iteration impacts the transactor performance: it uses bandwidth between SW and HW to transmit the status when the bandwidth for data ports decreases and impacts the `clk` frequency

7.3.3 Compiling and running

The `tb.cc` file has to be compiled in a dynamic library (`tb.so`) for **`zEmiRun`**:

```
$ g++ -fPIC -shared -o tb.so ../src/tb.cc –I../
  zcui.work/zebu.work/ -I$ZEBU_ROOT/include/ -L$ZEBU_ROOT/lib
  –lZebuZEMI3
```

The following command will launch the test:

```
$ zEmiRun -z ../ zcui.work/zebu.work -m ./tb.so:zemi3_main
```

7.4 Notifying to the SW the end of a HW process

7.4.1 Functional description and ZEMI-3 highlighted features

This example shows how the HW can notify the SW that an event occurs by using an import function.

This function has only inputs to transfer data from the DUT to the SW, therefore it is automatically processed in *streaming mode* by the ZEMI-3 infrastructure.

The SW side consists in the implementation of the import function.

The execution mode is the stand-alone mode.

7.4.2 Implementation

7.4.2.1 Architecture

In this simple testcase the import function is called when the `process_end` signal coming from the DUT has a posedge. When the posedge is detected, the import function is called with two parameters: a controlled clock cycle counter and a 32-bit data coming from the DUT.

The SW part of the transactor implements the import function but there is no need for a testbench function. The import function increments an event counter then saves the cycle at which the event occurred and the value of the data coming from the DUT.

The import function does not need to be defined as a context because it does not need to call an export and the scope information is not needed inside this function.

7.4.2.2 HW part of the transactor

The transactor module interface is the following:

- `clk`: controlled clock
- `process_end`: DUT output port to notify the end of a process
- `data`: 32-bit data bus coming from the DUT

```
module transactor(input clk, input process_end, input [31:0] data);
```

The controlled clock cycle counter is declared and initialized in one operation:

```
  reg [31:0] counter = 0;
```

The import function is declared with its interface:

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

Detection of the `process_end` posedge and call of the import function:

```
  always @(posedge process_end)
    notify_process_end(counter, data);
```

Increment of the counter at each cycle of the controlled clock:

```
  always @(posedge clk)
    counter = counter + 1;

endmodule
```

7.4.2.3 DVE file

The DVE file corresponding to this test case is the following:

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

transactor xtor0 (
     .clk ( clk ),
     .process_end( process_end ),
     .data( data[31:0] )
);
```

7.4.2.4 SW part of the transactor

The SW part of the transactor consists in the implementation of the import function. This function is automatically called by the ZEMI-3 manager in **`zEmiRun`** when it is called from HW.

An `import.hh` header file declares the structure which saves the event information and the global variables:

```
#ifndef _import_hh_
#define _import_hh_

typedef struct event {
  unsigned int cycle;
  unsigned int data;
};

extern event *EVENTS;
extern unsigned int EVENTS_NB;

#endif
```

The `event` structure contains the cycle number at which the event occurs and the data read from the DUT.

Two global variables can be used by other functions: an array of events, `EVENTS`, and the size of this array, `EVENTS_NB`.

The `import.cc` file containing the implementation of the import function includes:

- The header file generated for the transactor which can be found in the `zebu.work` directory
- The `import.hh` file

The maximum number of events (`MAX_EVENTS`) is defined and it is used to allocate the `EVENTS` array.

```
#include <transactor.h>
#include <stdio.h>
#include "import.hh"

#define MAX_EVENTS 10

event *EVENTS = NULL;
unsigned int EVENTS_NB = 0;
```

The import function is declared as `extern "C"`. The arguments are of type `svBitVecVal` since the types in the SystemVerilog declaration were bits. They are declared as `const` because they cannot be changed in the import function.

```
extern "C" void notify_process_end(const svBitVecVal counter[1], const
svBitVecVal data[1])
{
```

If the call is the first call, the `EVENTS` array is allocated .The values received from the HW are displayed and saved in the `EVENTS` array. When the `MAX_EVENTS` is reached, the emulation terminates:

```
  if (EVENTS_NB==0) {
    EVENTS = new event[MAX_EVENTS];
  }
 printf("HW event notified at cycle '%d', data = %08x\n", counter[0],
data[0]);

  EVENTS[EVENTS_NB].cycle = counter[0];
  EVENTS[EVENTS_NB].data = data[0];
  EVENTS_NB++;

  if (EVENTS_NB == MAX_EVENTS) {
    printf("End of run\n");
    EVENTS_NB = 0;
    delete [] EVENTS;
    EVENTS=NULL;
    exit(0); // end of emulation
  }
}
```

7.4.3 Compiling and running

The file `import.cc` has to be compiled into a dynamic library (`import.so`) for **`zEmiRun`**:

```
$ g++ -rdynamic -fPIC -shared -o import.so ../src/import.cc
  –I../ zcui.work/zebu.work/ -I$ZEBU_ROOT/include/ -L$ZEBU_ROOT/lib
  –lZebuDZEMI3
```

Note: The `–rdynamic` and `–fPIC` options should be used when the test contains import functions. Without these options the `import.so` dynamic library should be linked to the transactor-generated dynamic library.

The following command launches the test:

```
$ zEmiRun -z ../ zcui.work/zebu.work -m ./import.so
```

Note: Since no testbench exists in this test case, then no testbench function is specified in the command line.

7.5 Simple SW memory model

7.5.1 Functional description and ZEMI-3 highlighted features

In this example, the transactor models a memory which is mapped on the PC and can be accessed from the DUT. Both READ and WRITE accesses are possible through the transactional interface and are initiated by the HW part of the transactor:

- The READ access is implemented by an import function with inputs (address) and outputs (data)
- The WRITE access is implemented by an import function with only inputs (address and data) which would be automatically switched to *streaming mode* by the ZEMI-3 infrastructure. However, the *streaming mode* has to be disabled in the present example because it may cause data corruption

Imagine that the DUT writes 100 addresses consecutively in 100 clock cycle, and at the 101th cycle performs a READ at the last address written. If the WRITE is in *streaming mode*, there is no guarantee that the last WRITE will be executed in SW at the time the READ is executed.

This example runs in stand-alone mode with **`zEmiRun`**.

7.5.2 Implementation

7.5.2.1 Architecture

The memory modeled with the transactor is a 16-bit address and 32-bit data memory (64 kB). Its interface has a write enable high (`we`), a read enable high (`re`), a clock port (`clk`) and two data ports (`din` for data in and `dout` for data out). It is a read-after-write memory.

The import functions `readData` and `writeData`. The *streaming mode* is disabled for the `writeData` function to avoid data corruption.

Both functions are declared as `context` functions since the memory transactor may be instantiated several times in the same environment.

In the HW part of the transactor, the import functions are called when the appropriate enable logic is `true`. In the SW part of the transactor, the data context is switched according to the scope information provided in each call.

An initialize method of the `Memory32` class is used to initialize the SW part of the memory.

7.5.2.2 HW part of the transactor

The interface of the transactor is the following:

```
module mem_xtor(input clk,
                input [15:0] addr,
                input [31:0] din,
                input we,
                input re,
                output reg [31:0] dout);
```

The declaration of the READ function is the following:

```
import "DPI-C" context function void readData(input bit [15:0] addr,
output bit [31:0] dout);
```

The declaration of the WRITE function includes the attribute which disables the *streaming mode*:

```
(* zebu_stream = 0 *)
import "DPI-C" context function void writeData(input bit [15:0] addr,
input bit [31:0] din);
```

The model of the memory is the following:

```
  initial dout = 0;
// memory behavior
  always @(posedge clk)
    if (we) begin
      writeData(addr, din);
      if (re)
        dout = din;
    end
    else
      if (re) readData(addr, dout);

endmodule
```

7.5.2.3 DVE file

The DVE file corresponding to this Lab is the following:

```
zceiClockPort c(.cclock(clk), .cresetn(rst));

mem_xtor mem_xtor0( .clk(clk),
                    .addr(mem_addr[15:0]),
                    .din(mem_din[31:0]),
                    .dout(mem_dout[31:0]),
                    .we(mem_we),
                    .re(mem_re)
                  );
```

7.5.2.4 SW part of the transactor

The SW part of the transactor consists in:

- a C++ class which represents the memory
- the implementation of the import functions

The Memory32 class represents a 32-bit wide memory with configurable depth. This class reads from and writes to the memory and displays warning messages when an Uninitialized Memory Read (UMR) is detected.

The Memory32 class is declared in the Memory32.hh file:

```
#ifndef _Memory32_hh_
#define _Memory32_hh_

// 32 bits wide memory
// with check of UMR
class Memory32
{
  public:
    Memory32(unsigned int depth);
    ~Memory32();

    unsigned int readData(const unsigned int addr);
    void writeData(const unsigned int addr, const unsigned int din);

  private:
    unsigned int *_memArray;
    bool         *_accArray;
};
#endif
```

The Memory32 class is implemented in the Memory32.cc file:

```
#include "Memory32.hh"

Memory32::Memory32(unsigned int depth)
{
  _memArray = new unsi(gned int [depth];
  _accArray = new bool  [depth];
  memset(_memArray, 0, depth*sizeof(uint));
  memset(_accArray, false, depth*sizeof(bool));
}

Memory32::~Memory32()
{
  delete [] _memArray;
  delete [] _accArray;
}

unsigned int Memory32::readData(const unsigned int addr)
{
  if (_accArray[addr] == false)
    printf("# Warning: UMR detected at address '%d'", addr);
  return _memArray[addr];
}

void Memory32::writeData(const unsigned int addr, const unsigned int din)
{
  _accArray[addr] = true;
  _memArray[addr] = din;
}
```

The context imports are implemented in the `xtor.cc` file:

The `initialize` function is called at `init` time. This function allocates the memory object and registers user data. The `svPutUserData` function registers a `Memory32` object address. The values of the `key` parameters in the `svPutUserData` functions are the addresses of the import functions. The `initialize` implementation is:

```
extern "C" void initialize()
{
  printf("# Initializing SW part of the transactor\n");
  svScope s = svGetScope();
  Memory32 *newMem = new Memory32(MEM_DEPTH);
  svPutUserData(s, (void *)(readData), (void *)(newMem));
  svPutUserData(s, (void *)(writeData), (void *)(newMem));
}
```

The `readData` import function uses `svGetUserData` to get the `Memory32` object registered in the `initialize` function:

```
extern "C" void readData(const svBitVecVal addr[1], svBitVecVal dout[1])
{
  svScope s = svGetScope();
  Memory32 *mem = (Memory32 *)(svGetUserData(s, (void *)(readData)));
  dout[0]  = mem->readData(addr[0]);
}
```

Just like `readData`, the `writeData` import function uses `svGetUserData` to get the `Memory32` object:

```
extern "C" void writeData(const svBitVecVal addr[1], const svBitVecVal
din[1])
{
  svScope s = svGetScope();
  Memory32 *mem = (Memory32 *)(svGetUserData(s, (void *)(writeData)));
  mem->writeData(addr[0], din[0]);
}
```

7.5.3 Compiling and running

The `xtor.cc` and `Memory32.cc` files have to be compiled into a dynamic library (`xtor.so`) for **`zEmiRun`**:

```
$ g++ -rdynamic -fPIC -shared -o xtor.so ../src/xtor.cc
  ../src/Memory32.cc –I../ zcui.work/zebu.work/
  -I$ZEBU_ROOT/include/ -L$ZEBU_ROOT/lib –lZebuZEMI3
```

Note: `–rdynamic` option should be used when the test contains import functions. Without this option, `import.so` should be linked with the dynamic library of the transactor in the `zebu.work` directory: `zcui.work/zebu.work/xtor.so`

The following command launches the test:

```
$ zEmiRun -z ../ zcui.work/zebu.work -m ./xtor.so
```

Note: Since no testbench exists in this test case, then no testbench function is specified in the command line.

7.6 Streaming data to the testbench

7.6.1 Functional description and ZEMI-3 highlighted features

In this Lab, data is streamed from HW to SW for trace purpose, without impacting the speed of the HW part.

The data exchange is initiated by the HW using an import function with only inputs since there is no expected data from the SW. ZEMI-3 automatically switches this function into *streaming mode* to get the best performance.

The SW part consists in the implementation of the import function with an additional tracer class which dumps data into a binary file.

7.6.2 Implementation

7.6.2.1 Architecture

The transactor traces the data bus in the following way:

- The HW part of the transactor sends the data of the bus at each posedge of the controlled clock if the bus is active.
- The SW part of the transactor writes the data into a file.

The transactor is instantiated twice to monitor two different buses of the DUT.

7.6.2.2 HW part of the transactor

The transactor module interface is the following:

```
module streamer_out(input clk, input [31:0] data_bus, input [31:0]
addr_bus, input bus_enabled);
```

Where:

- `clk` is the controlled clock
- `data_bus` input is the 32-bit data bus traced in this example
- `bus_enabled` signal; when it is high the transactor will send the `data_bus` value to the SW

Two imports are declared in the transactor:

- `initialize` import function is called in an `initial` block. It initializes the data structure according to the scope of the caller:
  ```
  import "DPI-C" context function void initialize();
  ```
- `stream_out` import function uses the cycle number of the call and the value of `data_bus`. It traces the data going to the SW:
  ```
  import "DPI-C" context function void stream_out(input bit [31:0]
  cycle, input bit [31:0] data, input bit [31:0] addr);
  ```

Declaration of the cycle counter and initialization of the transactor:

```
  reg [31:0] cycle;

  initial begin
    cycle <= 0;
    initialize();
  end
```

In the always block the stream_out function is called on posedge of clk when the bus is enabled:

```
  always @(posedge clk)
    if (bus_enabled)
      stream_out(cycle, data_bus, addr_bus);

  always @(posedge clk)
    cycle <= cycle + 1;
endmodule
```

7.6.2.3 DVE file

In the DVE file two instances s0 and s1 of the transactor are created.

```
zceiClockPort c(
  .cclock(clk)
);
streamer_out s0(.clk(clk),
                .data_bus(data_bus_0[31:0]),
                .addr_bus(addr_bus_0[31:0]),
                .bus_enabled(bus_0_enabled)
               );
streamer_out s1(.clk(clk),
                .data_bus(data_bus_1[31:0]),
                .addr_bus(addr_bus_1[31:0]),
                .bus_enabled(bus_1_enabled)
               );
```

7.6.2.4 SW part of the transactor

The SW part consists in the following:

- A tracer structure which traces data into a file
- The import functions implementation

The tracer structure includes the following methods:

- open: opens the trace file
- close: closes the trace file
- writeData: writes data into the file

The declaration of the tracer structure in the file tracer.hh is the following:

```
#ifndef _tracer_hh_
#define _tracer_hh_

#include <streamer_out.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <string>
using std::string;

#define NB_MAX_TRACE 100000

struct tracer {

  tracer(const char *name);
  void open();
  void write(unsigned int cycle, unsigned int data, unsigned int addr);
  void close();
  unsigned int samples() { return _nb; }
  ~tracer();

  const char  *_name;
  FILE        *_file;
  unsigned int _nb;
};
#endif
```

The tracer structure is implemented in the `tracer.cc` file:

```
#include "tracer.hh"

tracer::tracer(const char *name)
{
  _name = strdup(name);
  _file = NULL;
  _nb = 0;
}

tracer::~tracer()
{
  free((void *)_name);
}

void tracer::open()
{
  if (_file != NULL) {
    printf("ERROR: tracer '%s' is already opened.", _name);
    exit(1);
  }

  string tmp = _name;
  tmp += ".bin";
  _file = fopen(tmp.c_str(), "w+");

  if (_file == NULL) {
    printf("ERROR: Cannot open file '%s'\n", _name);
    exit(1);
  }
}

void tracer::write(unsigned int cycle, unsigned int data, unsigned int
addr)
{
  if (_file == NULL) {
    printf("ERROR: tracer '%s' is not initialized", _name);
    exit(1);
  }
  fwrite(&cycle, sizeof(uint), 1,_file);
  fwrite(&data, sizeof(uint), 1, _file);
  fwrite(&addr, sizeof(uint), 1, _file);

  _nb++;
}

void tracer::close()
{
  if (_file == NULL) {
    printf("ERROR: Cannot close tracer '%s'.", _name);
    exit(1);
  }
  fclose(_file);
  _file = NULL;
}
```

The `xtor.cc` file implements the import functions and calls the `tracer` structure.

The inputs are declared as `context` and SystemVerilog functions `svGetUserData` and `svPutUserData` are used to register the handler of the corresponding tracer.

The `initialize` function creates the tracer objects and initializes them with the appropriate names.

```
extern "C" void initialize()
{
  svScope s = svGetScope();

  // Creating the data structure
  tracer *t = new tracer(svGetNameFromScope(s));
  t->open();

  // Registration of the tracer object
  svPutUserData(s, (void *)(stream_out), (void *)(t));
}
```

The `stream_out` import function writes the data into the dump file. The tracer object is obtained with the `svGetUserData` function.

Note: The emulation is stopped when the number of samples is equal to `NB_MAX_TRACE`.

```
extern "C" void stream_out(const svBitVecVal cycle[1], const svBitVecVal
data[1], const svBitVecVal addr[1])
{
  svScope s = svGetScope();
  tracer *t = (tracer *)(svGetUserData(s, (void *)(stream_out)));

  t->write(cycle[0], data[0], data[0]);

  // end of the test
  if (t->samples() == NB_MAX_TRACE) {
    t->close();
    delete t;
    exit(0);
  }
}
```

7.6.3 Compiling and running

```
$ g++ -rdynamic -fPIC -shared -o xtor.so ../src/tracer.cc
  ../src/xtor.cc -I../ zcui.work/zebu.work/ -I$(ZEBU_ROOT)/include/
  -L$(ZEBU_ROOT)/lib  -lZebuZEMI3
```

```
$ zEmiRun -z ../ zcui.work/zebu.work -m ./xtor.so
```

7.7 Streaming data to the testbench with a configurable address range

7.7.1 Functional description and ZEMI-3 highlighted features

This example is based on the previous one. We need in addition to trace only a specific range of addresses. This range of addresses has to be configurable.

In this case the filter has to be done in the HW side by simply adding an `if` statement before the call of the `stream_out` function. The modification has to be done on the HW side because processing is very simple, and because it will decrease the bandwidth for the `stream_out` function when the address is out of the trace range.

In a more generic manner, when the treatments are simple and can be simply synthesized, they have to be created in the HW side. When processing is more complex it must be done in the SW side.

7.7.2 Implementation

The modification of the previous example is the following:

- Add two outputs to the `initialize` import functions representing the range of addresses to trace.
- Add the comparison before the call of the `stream_out` function.

7.7.3 Modifications

In the HW part, the interface of the `initialize` import changes:

```
  import "DPI-C" context function void initialize(output bit [31:0] low,
output bit [31:0] high);
```

The following registers are added and used as arguments for the `initialize` method:

```
  reg [31:0] low_addr;
  reg [31:0] high_addr;
  initial begin
    cycle <= 0;
    initialize(low_addr, high_addr);
  end
```

A test is added in the `always` block which calls the `stream_out` function:

```
  always @(posedge clk)
    if (bus_enabled)
      if ((addr_bus >= low_addr) && (addr_bus <= high_addr))
        stream_out(cycle, data_bus, addr_bus);
```

In the SW side, the two arguments are added to the import function interface `initialize`, and their values are set in the function:

```
extern "C" void initialize(svBitVecVal low[1], svBitVecVal high[1])
{
  svScope s = svGetScope();

  // Creating the data structure
  tracer *t = new tracer(svGetNameFromScope(s));
  t->open();

  // Registration of the tracer object
  svPutUserData(s, (void *)(stream_out), (void *)(t));

  low[0]  = 0x00000100;
  high[0] = 0x00001000;
}
```

7.8 Streaming data to the design

7.8.1 Functional description and ZEMI-3 highlighted features

The problematic of this example is how to stream data to the DUT from the SW. If data is available to the DUT, a signal `data_valid` becomes high. If no data is available to the DUT, the signal `data_valid` is low. The DUT takes the data on posedge of the controlled clock if `data_valid` is high.

Since the SW is the master of the exchange, an export will be used.

The data stream is unidirectional so the export will have only inputs.

It will be implemented as a task because it needs to consume simulation time.

7.8.2 ZEMI-3 specific

When an export task has only inputs, it is automatically declared in *streaming mode*. It means that the SW will not wait until the HW has actually executed the task before continuing. It increases the performance of the transactor.

When no export task is executing, the clocks are running by default. In this specific transactor, when the export task is not executing, the clocks are running and the `dut_valid` signal is low. In some applications, the stream has to be continuous. It means that no extra clock cycles are supported between the executions of the export taks. With the ZEMI-3 *back-to-back mode*, the clocks are automatically stopped when no export task is executing. This mode can be enabled at compilation and/or at runtime.

7.8.3 Implementation

7.8.3.1 Architecture

The transactor consists in a single export task with a 32-bit input. When this task is called, it puts the data received on the DUT bus and asserts the `data_valid` signal. It then waits for a `clk` posedge, de-asserts the `data_valid` signal then waits for a `clk` negedge.

7.8.3.2 HW part of the transactor

The transactor module interface is the following:

```
module streamer_in(input clk, output reg [31:0] data, output reg
data_valid);
```

Where:

`clk` is the controlled clock
`data` is connected to the DUT bus
`data_valid` is the signal telling the DUT that data can be taken.

The declaration of the export task without its arguments:
```
  export "DPI-C" task stream_in;
```

Initialization of the outputs:
```
initial data_valid = 0;
initial data = 32'hD000000D;
```

Implementation of the streaming task:
```
  task stream_in(input bit [31:0] d);
    begin
      data = d;
      data_valid = 1;
      @(posedge clk);
      data_valid = 0;
      @(negedge clk);
    end
  endtask
```

7.8.3.3 DVE file

The instantiation of the transactor in the DVE file is the following:
```
zceiClockPort c(.cclock(clk));
streamer_in s0( .clk(clk),
                .data(data[31:0]),
                .data_valid(data_valid)
              );
```

7.8.3.4 SW part of the transactor

The SW part consists in the implementation of the testbench function provided to **`zEmiRun`**. This function sets the transactor scope and performs a loop in which it calls the streaming export task.
```
#include "streamer_in.h"

extern "C" int bench()
{
  svScope s = svGetScopeFromName("s0");
  svSetScope(s);

  for(unsigned int i=0;i<1000000;i++) {
    stream_in(&i);
  }
  return 0;
}
```

7.8.4 Compiling and running

```
$ g++ -rdynamic -fPIC -shared -o tb.so ../src/tb.cc
  -I../zcui.work/zebu.work/ -I$(ZEBU_ROOT)/include/
  -L$(ZEBU_ROOT)/lib  -lZebuZEMI3

$ zEmiRun -z ../zcui.work/zebu.work -m ./tb.so
```

8 Reference Resources

8.1 ZeBu-UF documents

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

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

[0] The *ZeBu-UF Product Overview* describes the different ZeBu-UF products and their features.

[1] The *ZeBu-UF Installation Manual* describes how to install the ZeBu-UF SW and HW.

[2] The *ZeBu-UF Compilation Manual* describes the ZeBu-UF compilation process.

[3] The *ZeBu HDL Co-simulation Manual* describes the use of the HDL co-simulation driver for the ZeBu systems.

[4] The *ZeBu C++ Co-simulation Manual* describes the use of the C++ co-simulation driver for the ZeBu systems.

[5] The *ZeBu SystemC Co-simulation Manual* describes the use of the SystemC co-simulation driver for the ZeBu systems.

[6] The *ZeBu Synthesizable Test Bench Manual* describes the use of a Synthesizable Test Bench to drive a target design mapped in ZeBu interface FPGAs.

[7] The *ZeBu Transaction-Based Verification Manual* describes the use of the transaction-based mode for the ZeBu systems.

[8] The *ZeBu zRun Emulation Interface Manual* describes the **zRun** emulation control interface and how to use the different functions.

[9] The *ZeBu-UF Tutorial* illustrates, via a set of examples, how to use the ZeBu-UF platform for verifying a design-under-test.

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

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

[12] The *ZeBu-UF Reference Manual* provides detailed information on program commands, memory models, libraries, and files necessary to compile and verify a design-under-test.

[13] The *ZeBu C API Reference Manual* and *ZeBu C++ API Reference Manual* provide detailed information on C/C++ library, files, and interfaces necessary to write a C/C++ test bench to verify your design.

[14] The *ZEMI-3 Manual* introduces the ZEMI-3 infrastructure together with the advantages of transaction-based verification. It presents ZEMI-3 features and gives elements to choose the most appropriate architecture for your transactor with recommendations to write the HW and SW parts of your transactor.

8.2 ZeBu-XXL documents

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

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

[0] The *ZeBu-XXL Product Overview* describes the different ZeBu-XXL products and their features.

[1] The *ZeBu-XXL Installation Manual* describes how to install the ZeBu-XXL SW and HW.

[2] The *ZeBu-XXL Compilation Manual* describes the ZeBu-XXL compilation process.

[3] The *ZeBu HDL Co-simulation Manual* describes the use of the HDL co-simulation driver for the ZeBu systems.

[4] The *ZeBu C++ Co-simulation Manual* describes the use of the C++ co-simulation driver for the ZeBu systems.

[5] The *ZeBu SystemC Co-simulation Manual* describes the use of the SystemC co-simulation driver for the ZeBu systems.

[6] The *ZeBu Synthesizable Test Bench Manual* describes the use of a Synthesizable Test Bench to drive a target design mapped in ZeBu interface FPGAs.

[7] The *ZeBu Transaction-Based Verification Manual* describes the use of the transaction-based mode for the ZeBu systems.

[8] The *ZeBu zRun Emulation Interface Manual* describes the **zRun** emulation control interface and how to use the different functions.

[9] The *ZeBu Tutorial* illustrates, via a set of examples, how to use the ZeBu-UF platform for verifying a design-under-test.

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

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

[12] The *ZeBu-XXL Reference Manual* provides detailed information on program commands, memory models, libraries, and files necessary to compile and verify a design-under-test.

[13] The *ZeBu C API Reference Manual* and *ZeBu C++ API Reference Manual* provide detailed information on C/C++ library, files, and interfaces necessary to write a C/C++ test bench to verify your design.

[14] The *ZEMI-3 Manual* introduces the ZEMI-3 infrastructure together with the advantages of transaction-based verification. It presents ZEMI-3 features and gives elements to choose the most appropriate architecture for your transactor with recommendations to write the HW and SW parts of your transactor.

8.3 ZeBu-Server Document Package

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

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

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

[2] The ***ZeBu-Server Compilation Manual*** describes the compilation process.

[3] The ***ZeBu HDL Co-simulation Manual*** describes the use of the HDL co-simulation driver for the ZeBu systems.

[4] The ***ZeBu C++ Co-simulation Manual*** describes the use of the C++ co-simulation driver for the ZeBu systems.

[8] The ***ZeBu zRun Emulation Interface Manual*** describes the zRun emulation control interface and how to use the different functions.

[13] The ***ZeBu C API Reference Manual*** and ***ZeBu C++ API Reference Manual*** provide detailed information on C/C++ library, files, and interfaces necessary to write a C/C++ test bench to verify your design.

[14] The ***ZEMI-3 Manual*** introduces the ZEMI-3 infrastructure together with the advantages of transaction-based verification. It presents ZEMI-3 features and gives elements to choose the most appropriate architecture for your transactor with recommendations to write the HW and SW parts of your transactor.

9 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	14F1, No 371, Sec. 1 Guangfu Rd., East District Hsinchu City 300 TAIWAN (R.O.C.) Tel: +886-(3)-564-7900