

Riviera™ – SystemC co-simulation (Windows)

Required Software:

Riviera 2003.03 build 846
Sample Project and Interface
SystemC 2.01
Microsoft Visual C++

<http://www.aldec.com/downloads>
<http://www.aldec.com/riviera/systemc.zip>
<http://www.systemc.org/projects/systemc>
Obtained from Microsoft

Overview

Riviera 2003.03 now supports the ability to co-simulate SystemC with VHDL, Verilog and Mixed HDL. Co-simulation is especially useful for the growing number of system-level design starts incorporating C components and testbenches. Results of both RTL and C can be viewed in the Riviera Waveform Viewer/Editor. The following document describes how to connect Riviera 2003.03 for Windows to co-simulate a Verilog design with SystemC. Before you begin make sure to obtain the above software required for this application note.

Included in this document:

Building PLI Applications
Building VHPI Application
Running the Sample SystemC Design
Known Limitations

Tips Before you begin

1. Unzip project archive preserving directory structure. Library `sc_interface_pli.dll` necessary for project simulation is located in PLI subfolder.
2. If you make any changes to C++ project files, you will have to rebuild this library: Go to the PLI subfolder and open `MAKEFILE.PLI` file using notepad (or any text editor).
3. If necessary, modify the following entries to match your directory structure:
 - `RIVIERA_INSTALL_DIR=<Install Directory>\Program files\Aldec\Riviera-2003.03"`
 - `SC_INSTALL_DIR=<Install Directory>\SystemC\systemc-2.0.1"`
 - `$(SC_INSTALL_DIR)\msvc60\systemc\Debug\systemc.lib`
 - Save your changes.
 - From the PLI subfolder, start `make_sc_interface.bat` to create `sc_interface_pli.dll`."

Building PLI Applications

Riviera loads specified PLI applications dynamically when the simulation process is initialized. Therefore, PLI applications must be compiled and linked to a shared library. PLI applications are loaded using operating system calls so the build process is different on every platform.

Header Files

The `pli/interface` subdirectory of the Riviera installation directory contains header files that must be included in your PLI application.

```
#include <veriusr.h>
#include <aldecpli.h>
```

The `aldecpli.h` header file contains an include directive for the `acc_user.h` header file. PLI applications must be registered through an array of `s_tfc` structures. The structure is defined in the `aldecpli.h` file.

```
typedef struct t_tfc
{
    short    type;
```

```

int    data;
int    (*checktf)();
int    (*sizetf)();
int    (*calltf)();
int    (*misctf)();
char*   tfname;
/* The following fields are ignored */
int    forwref;
char    *tfveritool;
char    *tferrmessage;
int    hash;
struct  t_tfcell *left_p;
struct  t_tfcell *right_p;
char    *namecell_p;
int    warning_printed;
} s_tfcell, *p_tfcell;

```

Compiling and Linking

To compile your PLI application using the Visual Studio, you need to add the `aldecli.lib` to the Resource Files folder. The `aldecli.lib` file is located in the `pli/lib` subdirectory of the Riviera installation directory. Next, add a definition file to the Header Files folder. The definition file should contain the following entry:

```

EXPORTS
veriusertfs DATA

```

Then we can compile the C/C++ files in Visual Studio.

Sample PLI Application

This section presents a sample PLI application that can be used to print the structure of the design to the Console in the text mode.

The application is registered in the `veriusertfs` array. The entry in the array registers a `usertask`. The task is registered as a `calltf` routine. Every time Riviera encounters a `$structure` task in the HDL code, it will execute the `structure` function. The `structure()` function calls `printNextBranch()` that scans recursively the structure of the design. Handles to subsequent modules are obtained with `acc_next_child()`. Once you have the module handle, you can get the name of the module with `acc_fetch_fullname()`.

```

#include "veriusert.h"
#include "aldecli.h"
#include "acc_user.h"

int level=0;

void printNextBranch(handle module_handle)
{
    handle child_handle = null;
    // scan recursively beginning with top-level modules
    while (child_handle = acc_next_child(module_handle, child_handle))
    {
        io_printf("%s\n", acc_fetch_fullname(child_handle));
        printNextBranch(child_handle);
    }
}

int structure()
{
    // initialize environment for access routines
    acc_initialize();
    printNextBranch(null);
    acc_close();
    return (0);
}

```

```
s_tfcell veriusertfs[] =
{
  {usertask, 0, 0, 0, structure, 0, "$structure", 0},
  {0}
};
```

The usertask **\$structure** can be called from the Verilog source code, like any other system task or function:

```
initial
  $structure
```

After initializing the simulation process, you can also call Verilog tasks interactively from the Console. To print the structure of the design to the Console, simply type **\$structure** at simulator prompt.

NOTE: It is not possible to pass arguments to tasks and functions called from the Console.

Building VHPI Applications

VHPI applications must be compiled and linked to a shared library. They are registered through VHDL foreign architectures at simulation startup.

Header Files

The **pli/interface** subdirectory of the Riviera installation directory contains header files that must be included in your PLI application.

```
#include <vhpi_user.h>
#include <aldecpli.h>
```

The **aldecpli.h** header file contains an include directive for the **acc_user.h** header file. The **aldecpli.lib** file needed to link VHPI libraries under Windows is located in the **pli/lib** subdirectory.

Registering VHPI Applications

Each VHPI library must include at least one function to register VHPI tasks. A pointer to this function is located in the **vhpi_startup_routines** array. The array must be null terminated. Names of the registering functions are arbitrary, for example:

```
PLI_VOID (*vhpi_startup_routines[])() = {
  startup_1,
  startup_2,
  null
};
```

Functions that register VHPI tasks, for example **startup_1()**, use the standard VHPI function **vhpi_register_foreignf()**. The **vhpi_register_foreignf()** function must be passed a pointer to the **vhpiForeignDataT** structure, for example:

```
PLI_VOID startup_1() {
  vhpiForeignDataT foreignDatap = {
    vhpiArchFK,
    "libvhpi",
    "myvhpitask",
    elabf_1,
    execf_1
  };
  vhpi_register_foreignf(&foreignDatap);
}
```

The **vhpiForeignDataT** structure is defined in the **vhpi_user.h** header file. It includes five elements:

vhpiArchFK

This constant specifies that the foreign model is an architecture. Registering VHPI applications through foreign procedures and foreign functions is supported in Riviera 2002.09 and newer. The following constants can be used depending on the foreign model:

vhpiArchFK specifies that the foreign model is an architecture

vhpiProcFK specifies that the foreign model is a procedure

vhpiFuncFK specifies that the foreign model is a function

char *libraryName

The name of the library containing VHPI routines. The name can be either absolute or relative to the Riviera current directory. If you omit the extension, Riviera will automatically add the default **.dll** extension. If the library cannot be found in the current directory, Riviera will check the locations pointed by the **PATH** system variable.

char *modelName

The name of the VHPI task that appears in the VHDL foreign attribute string. See [Foreign Architectures](#) and [Foreign Subprograms](#) below for information on the foreign attribute string in a VHPI foreign model.

void (*elabf) (const vhpiCbDataT* cbdata)

Pointer to the callback function for the elaboration of the foreign architecture. This function can only reference objects within the instance of the architecture that called the function. The function is executed during the elaboration of the VHDL code.

void (*execf) (const vhpiCbDataT* cbdata);

Pointer to the callback function for initialization of the foreign architecture. This function can access the whole VHDL design. The function is registered to occur when a foreign architecture, procedure or function is executed during simulation of VHDL design.

The callback function associated with the task could be defined as follows:

```
void elabf_1(const struct vhpiCbDataS* cb_p)
{
    vhpi_printf(
        "CALLING SCOPE NAME: %s",
        vhpi_get_str(vhpiNameP, cb_p->obj)
    );
}
```

Note that the function body can be empty:

```
void execf_1(const struct vhpiCbDataS* cb_p)
{
}
```

You can also pass a null pointer instead of a function.

Foreign Architectures

VHPI applications can be registered either through a foreign model, that is a foreign architecture or a foreign subprogram. This section describes registration through a foreign architecture.

A foreign architecture contains the **FOREIGN** attribute. The attribute is declared in the **std** library in the **standard** package. The string in a foreign architecture follows VHPI standard syntax:

```
attribute FOREIGN of <architecture_name>: architecture is
"VHPI <library_name>; <model_name>"
```

The foreign string includes three parts:

VHPI

The VHPI identifier indicates that the foreign model has a VHPI based implementation

<library name>

The name of the library containing VHPI routines. The name can be either absolute or relative to the Riviera current directory. If you omit the extension, Riviera will automatically add the default **.so (Unix, Linux)** or **.dll (Windows)** extension. If the library cannot be found in the current directory, Riviera will check the locations pointed by the **LD_LIBRARY_PATH (Unix, Linux)** or **PATH (Windows)** system variable.

<model_name>

Identifies a VHPI based model implementation for a foreign architecture. The name of the string must match *char *modelName* in the **vhpiForeignDataT** structure.

The foreign architecture could be defined as follows:

```
architecture ar_vhpi of en_vhpi is
  attribute foreign of ar_vhpi: architecture is
    "VHPI libvhpi; myvhpitask";

begin
end architecture ar_vhpi;
```

Foreign Subprograms

VHPI applications can be registered either through a foreign model, that is a foreign architecture or a foreign subprogram. This section describes registration through foreign subprograms.

A foreign subprogram is defined with the **FOREIGN** attribute. The attribute is declared in the **std** library in the **standard** package. The string in a foreign subprogram follows VHPI standard syntax:

```
procedure <procedure_name> is
begin
end procedure;

attribute foreign of <procedure_name>: procedure is
"VHPI <library_name>; <model_name>";
```

The string includes three parts:

VHPI

The VHPI identifier indicates that the foreign model has a VHPI based implementation.

<library name>

The name of the library containing VHPI routines. The name can be either absolute or relative to the Riviera current directory. If you omit the extension, Riviera will automatically add the default **.dll** extension. If the library cannot be found in the current directory, Riviera will check the locations pointed by the **PATH** system variable.

<model_name>

Identifies a VHPI based model implementation for a foreign architecture. The name of the string must match *char *modelName* in the **vhpiForeignDataT** structure.

The foreign procedure could be defined as follows:

```
architecture archVHPI of entVHPI is

procedure myVhpiTask is
begin
```

```

end procedure;

attribute foreign of myVhpiTask: procedure is
"VHPI libvhpi; myVhpiTask";

begin
-- your VHDL code
end architecture archVHPI;

```

NOTE: Some PLI and VHPI examples are included with Riviera. They are located in <Riviera Install Subdirectory>\Projects

PLI	VHPI
-----	-----
pli_mem1	vhpsample1
plitutorial	vhpsample2
	vhpitutorial

Setting SystemC Environment

Before compiling the sample project it is important to download the latest version of SystemC (currently 2.0.1) distribution from the link above, install it and compile using Microsoft Visual C++. Follow the instructions in the <Install Directory>\SystemC\systemc-2.0.1\msvc60\INSTALL file of SystemC installation.

Note: This step is optional if you do not plan to modify the design and only want to examine the sample.

Running the Sample SystemC Design

Once the interface is complete the following is an example showing how we can interface SystemC with Riviera. The sample consists of a SystemC counter and Verilog testbench. The Co-Simulation between HDL simulator and SystemC is possible thanks to I/O objects:

```

INVECTOR      invector;
OUTVECTOR     outvector;

```

Connection between HDL simulator and SystemC is done using PLI interface. All the necessary sources are included. The Verilog testbench instantiates a SystemC module:

```

-----testbench.v-----
module testbench;
  reg clock;
  reg clear;
  reg [0:7] dout;

  initial begin

    $monitor("%t %b %b", $time, clock, clear, dout);
    $sc_counter(clock, clear, dout);

    fork
      #0   clock = 0;
      #10  clock = 1;
      #20  clock = 0;
      #30  clock = 1; clear = 1;
      #40  clock = 0;
      #50  clock = 1; clear = 0;
      #60  clock = 0;
      #70  clock = 1;
      #80  clock = 0;
      #90  clock = 1;
      #100 clock = 0;
    endfork
  endmodule

```

```

        #110 clock = 1;
        #120 $finish;
    join

end
endmodule

```

-----end of testbench.v-----

The SystemC module (counter) is described in two files: `sc_counter.h` and `sc_counter.cpp`

-----sc_counter.h-----

```

#include "systemc.h"

SC_MODULE(counter)
{
    sc_in<bool> clock;
    sc_in<bool> clear;
    sc_out<sc_int<8> > dout;

    int countval;
    void onetwothree();

    SC_CTOR(counter)
    {
        SC_METHOD(onetwothree);
        sensitive_pos (clock);
    }
};

```

-----end of sc_counter.h-----

-----sc_counter.cpp-----

```

#include "sc_counter.h"
#include "sc_interface.h"

counter iCounter("iCounter");

sc_signal<bool>    clock;
sc_signal<bool>    clear;
sc_signal<sc_int<8> > dout;

extern "C" {
void counter_sc_calltf();
}

void counter_sc_calltf()
{
    iCounter.clock(clock);
    iCounter.clear(clear);
    iCounter.dout(dout);

    sc_trace_file *tf = sc_create_vcd_trace_file("sc_trace");
    sc_trace(tf, clock, "clock");
    sc_trace(tf, clear, "clear");
    sc_trace(tf, dout, "dout");

    sc_initialize();
}

extern "C" {
void counter_sc_misctf(void *invector, void *outvector, unsigned long simFor);
}

```

```

}

void counter_sc_misctf(void *invector, void *outvector, unsigned long simFor)
{
    INVECTOR *pInvector = (INVECTOR *)invector;

    clock = pInvector->clock ? true : false;
    clear = pInvector->clear ? true : false;

    sc_cycle(simFor);
    _flushall();

    OUTVECTOR *pOutvector = (OUTVECTOR *)outvector;

    pOutvector->dout = dout.read();
}

```

```

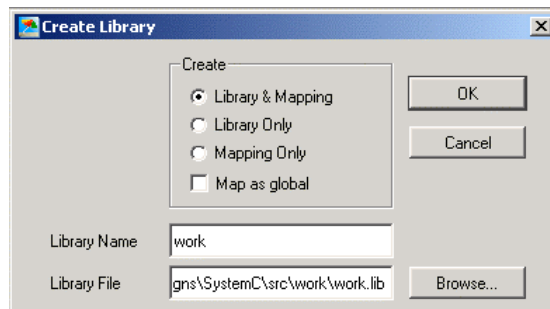
void counter::onetwothree()
{
    if (clear)
    {
        countval = 0;
    }
    else
    {
        countval++;
    }

    dout = countval;
}

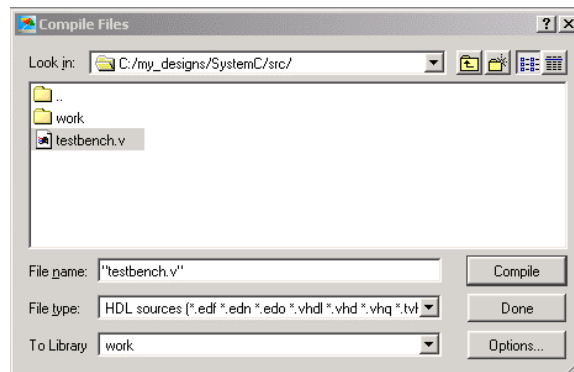
```

-----end of sc_counter.cpp-----

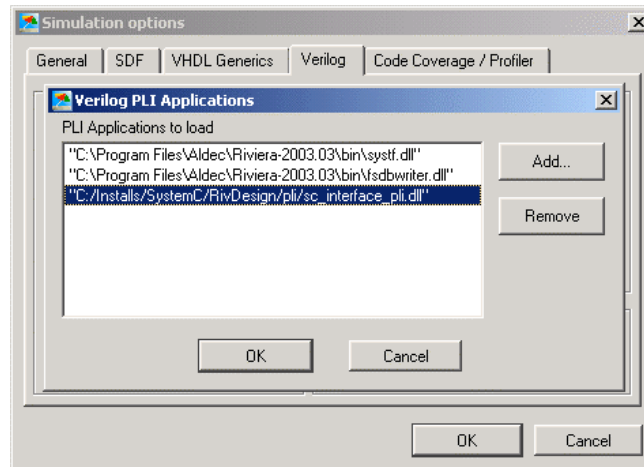
The SystemC-HDL Co-Simulation process can be run after initializing the simulation. Create a library called "work".



Compile top-level Verilog file (.../src/testbench.v)

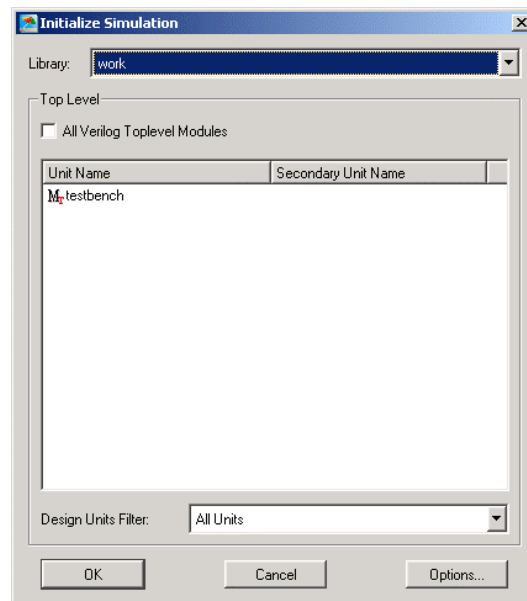


Go to Tools/ Simulation Options and select Verilog tab, click "PLI Applications".

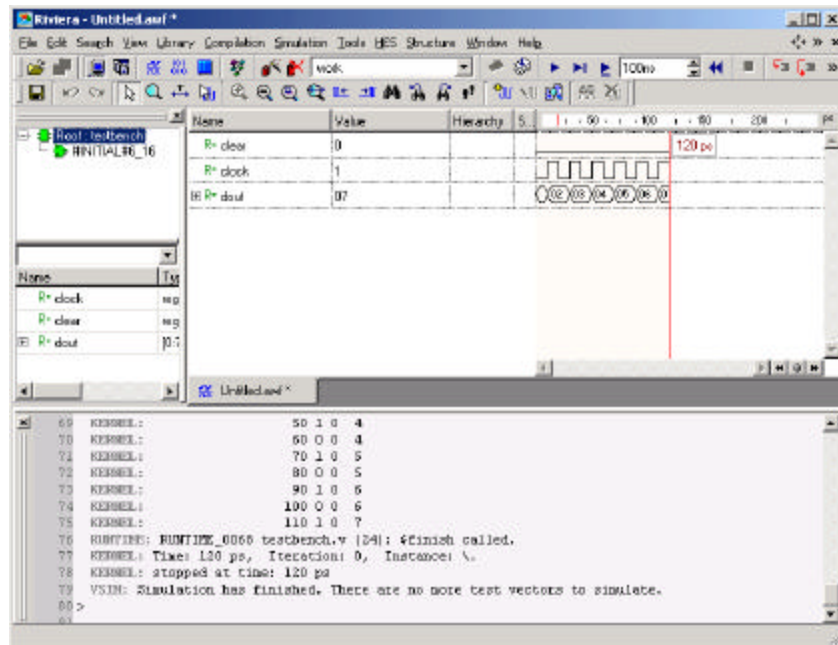


Click Add and point to the sc_interface_pli.dll file. Click Open, then OK and OK again.

Initialize simulation, selecting "testbench" as top level.



Create waveform window and select testbench signals for observation. Run simulation for 120 ps.



Known limitations:

- Riviera closes after ending simulation
- Result of output function (cout <<) is not displayed in the Riviera console window