GHDL Cosimulation Documentation

Release latest

Tristan Gingold and contributors

Apr 19, 2020

VHPIDIRECT

This repository contains documentation and working examples about how to co-simulate VHDL and other languages through GHDL's foreign interfaces. Since specific features of the language and the tool are used, it is suggested for users who are new to either *GHDL* or *VHDL* to first read the Quick Start Guide in the main documentation (ghdl.rtfd.io).

Three main approaches are used to co-simulate (co-execute) VHDL sources along with software applications written in a language other than VHDL (typically C/C++/SystemC):

- Verilog Procedural Interface (VPI), also known as Program Language Interface (PLI) 2.0.
- VHDL Procedural Interface (VHPI), or specific implementations, such as Foreign Language Interface (FLI).
- Generation of C/C++ models/sources through a transpiler.

VPI and VHPI are complex APIs which allow to inspect the hierarchy, set callbacks and/or assign signals. Because provided features are similar, GHDL supports VPI only. Furthermore, as an easier to use alternative, GHDL features a custom coexecution procedure named VHPIDIRECT, similar to SystemVerilog's Direct Programming Interface (DPI). As of today, generation of C++/SystemC models à la Verilator is not supported. However, a *vhdlator*/*ghdlator* might be available in the future.

CHAPTER 1

Type declarations

Only subprograms (**functions** or **procedures**) can be imported using the foreign attribute. In the following minimal *example*, the *sin* function is imported:

```
package math is
  function sin (v : real) return real;
  attribute foreign of sin : function is "VHPIDIRECT sin";
end math;

package body math is
  function sin (v : real) return real is
  begin
    assert false severity failure;
  end sin;
end math;
```

Requirements:

- A subprogram is made foreign if the `foreign` attribute decorates it.
- The `attribute` specification must be in the same declarative part as the subprogram and must be after it. This is a general rule for specifications.
- The value of the specification must be a locally static string.
- The value of the attribute must start with `VHPIDIRECT` (an upper-case keyword followed by one or more blanks). The linkage name of the subprogram follows. The path to a shared library can be optionally specified between the keyword and the name of the subprogram (see *shlib*).
- Even when a subprogram is foreign, its body must be present in VHDL. However, since it won't be called, you can make it empty or simply put an assertion. If the body is ever executed, that will mean that the foreign resource was not properly linked.
- Except for resources in the standard C library (which is linked by default), the object file with the source code for foreign subprogram(s) must then be linked to GHDL, expanded upon in *Linking object files*.

Note: Attribute `foreign` is declared in the 1993 revision of the `std.standard` package. Therefore, it cannot be used in VHDL 1987.

1.1 Restrictions on type declarations

Any subprogram can be imported. GHDL puts no restrictions on foreign subprograms. However, the representation of a type or of an interface in a foreign language may be obscure. Most non-composite types are easily imported:

integer types 32 bit word. Generally, *int* for *C* or *Integer* for *Ada*.

physical types 64 bit word. Generally, *long long* for *C* or *Long_Long_Integer* for *Ada*.

floating point types 64 bit floating point word. Generally, *double* for *C* or *Long_Float* for *Ada*.

enumeration types 8 bit word, or, if the number of literals is greater than 256, by a 32 bit word. There is no corresponding C type, since arguments are not promoted.

Non-composite types are passed by value. For the *in* mode (default), this corresponds to the *C* or *Ada* mechanism. *out* and *inout* interfaces are gathered in a record and this record is passed by reference as the first argument to the subprogram. As a consequence, it is not suggested to use *out* and/or *inout* modes in foreign subprograms, since they are not portable.

Composite types:

- Records are represented like a *C* structure and are passed by reference to subprograms.
- Arrays with static bounds are represented like a *C* array, whose length is the number of elements, and are passed by reference to subprograms.
- Unconstrained arrays are represented by a fat pointer. It is not suggested to use unconstrained arrays in foreign subprograms.
- Accesses to an unconstrained array are fat pointers. Other accesses correspond to an address/pointer and are passed to a subprogram like other non-composite types.
- Files are represented by a 32 bit word, which corresponds to an index in a table.

CHAPTER 2

Wrapping a simulation (ghdl_main)

You may run your design from an external program. You just have to call the `ghdl_main` function which can be defined:

in C:

```
extern int ghdl_main (int argc, char **argv);
```

in Ada:

```
with System;
...
function Ghdl_Main (Argc : Integer; Argv : System.Address)
  return Integer;
pragma import (C, Ghdl_Main, "ghdl_main");
```

Tip: Don't forget to list the object file(s) of this entry point and other foreign sources, as per *Linking foreign object files to GHDL*.

Attention: The `ghdl_main` function must be called once, since reseting/restarting the simulation runtime is not supported yet. A workaround is to build the simulation as a shared object and load the `ghdl_main` symbol from it (see *shghdl*).

Hint: Immitating the run time flags, such as `-gDEPTH=12` from `-gGENERIC`, requires the `argv` to have the executable's path at index 0, effectively shifting all other indicies along by 1. This can be taken from the 0 index of the `argv` passed to `main()`, or (not suggested, despite a lack of consequences) left empty/null.

Since `ghdl_main` is the entrypoint to the design (GRT runtime), the supported CLI options are the ones shown in Simulation (runtime). Options for analysis/elaboration are not required and will NOT work. See `-r`.

CHAPTER 3

Linking object files

3.1 Linking foreign object files to GHDL

You may add additional files or options during the link of *GHDL* using `-Wl,` as described in Passing options to other programs. For example:

```
ghdl -e -Wl,-lm math_tb
```

will create the `math_tb` executable with the `lm` (mathematical) library.

Note the `c` library is always linked with an executable.

Hint: The process for personal code is the same, provided the code is provided as a C source or compiled to an object file. Analysis must be made of the HDL files, then elaboration with `-e -Wl,personal.c [options...] primary_unit [secondary_unit]` as arguments. Additional C or object files are flagged as separate `-Wl,*` arguments. The elaboration step will compile the executable with the custom resources. Further reading (particularly about the backend particularities) is at Elaboration [-e] and Run [-r].

3.2 Linking GHDL object files to Ada/C

As explained previously in *Wrapping a simulation (ghdl_main)*, you can start a simulation from an *Ada* or *C* program. However the build process is not trivial: you have to elaborate your program and your *VHDL* design.

Hint: If the foreign language is C, this procedure is equivalent to the one described in *Linking foreign object files to GHDL*, which is easier. Thus, this procedure is explained for didactic purposes. When suitable, we suggest to use `-e`, instead of `--bind` and `--list-link`.

First, you have to analyze all your design files. In this example, we suppose there is only one design file, `design.vhdl`.

```
$ ghdl -a design.vhdl
```

Then, bind your design. In this example, we suppose the entity at the design apex is `design`.

```
$ ghdl --bind design
```

Finally, compile/bind your program and link it with your *VHDL* design:

in C:

```
gcc my_prog.c -Wl,`ghdl --list-link design`
```

in Ada:

```
$ gnatmake my_prog -largs `ghdl --list-link design`
```

See GCC/LLVM only commands for further details about `--bind` and `--list-link`.

CHAPTER 4

Dynamic loading

Building either foreign resources or the VHDL simulation model as shared libraries allows to decouple the build procedures.

4.1 Loading foreign objects from within a simulation

Instead of linking and building foreign objects along with GHDL, it is also possible to load foreign resources dynamically. In order to do so, provide the path and name of the shared library where the resource is to be loaded from. For example:

```
attribute foreign of get_rand: function is "VHPIDIRECT ./getrand.so get_rand";
```

4.2 Generating shared libraries

Tip: Ensure reading and understanding *Linking object files* before this one.

There are three possibilities to elaborate simulation models as shared libraries, instead of executable binaries:

- `ghdl -e -shared [options...] primary_unit [secondary_unit]`
- `ghdl -e -Wl,-shared -Wl,-Wl,--version-script=./file.ver -Wl,-Wl,-u, ghdl_main [options...] primary_unit [secondary_unit]`
- ``gcc -shared -Wl,`ghdl --list-link tb` -Wl,--version-script=./file.ver -Wl,-u,ghdl_main``

The only difference between the two later procedures is the entrypoint (GHDL or GCC). Preference depends on the additional options that users need to provide. The main difference with the former is that it will make all symbols visible in the resulting shared library. In the other two procedures, visible symbols will be the ones defined in the default `grt.ver` added by GHDL and the `file.ver` provided by the user. Note that `file.ver` must include `ghdl_main` and any other added by the user. See example *shghdl* and ghdl-cosim#2.

Hint: When GHDL is configured with `--default-pic` explicitly, it uses it implicitly when executing any `-a`, `-e` or `-r` command. Hence, it is not required to provide these arguments (fPIC/PIE) to GHDL. However, these might need to be provided when building C sources with GCC. Otherwise linker errors such as the following are produced:

Hint: For further details regarding how to call `ghdl_main` see *Wrapping a simulation (ghdl_main)*.

Note: Alternatively, if the shared library is built with `--bind` and `--list-link`, the output from the later can be filtered with tools such as `sed` in order to remove the default version script (accomplished in ghdl#640), and make all symbols visible by default. However, this procedure is only recommended in edge cases where other solutions don't fit.

4.3 Loading a simulation

Attention: By default, GHDL uses `grt.ver` to limit which symbols are exposed in the generated artifacts, and `ghdl_main` is not included. See *Generating shared libraries* for guidelines to generate shared objects with visible or filtered symbols.

In order to generate a position independent executable (PIE), be it an executable binary or a shared library, GHDL must be built with config option `--default-pic`. This will ensure that all the libraries and sources analyzed by GHDL generate position independent code (PIC).

PIE binaries can be loaded and executed from any language that supports C-alike signatures and types (C, C++, golang, Python, Rust, etc.). This allows seamless co-simulation using concurrent/parallel execution features available in each language: pthreads, goroutines/gochannels, multiprocessing/queues, etc. Moreover, it provides a mechanism to execute multiple GHDL simulations in parallel.

For example, in Python:

```python
import ctypes
gbin = ctypes.CDLL(bin_path)

args = ['-gGENA="value"', 'gGENB="value"']

xargs = (ctypes.POINTER(ctypes.c_char) * (len(args) + 1))()
for i, arg in enumerate(args):
    xargs[i] = ctypes.create_string_buffer(arg.encode('utf-8'))
return args[0], xargs

gbin.main(len(xargv)-1, xargv)

import _ctypes
# On GNU/Linux
_ctypes.dlclose(gbin._handle)
# On Windows
#_ctypes.FreeLibrary(gbin._handle)
```

See a complete example written in C in *shghdl*.

Tip: As explained in *Wrapping a simulation (ghdl_main)*, `ghdl_main` must be called once, since reseting/restarting the simulation runtime is not supported yet (see ghdl#1184). When it is loaded dynamically, this means that the binary file/library needs to be unloaded from memory and loaded again (as in *shghdl*).

Tip: See ghdl#803 for details about expected differences in the exit codes, depending on the version of the VHDL standard that is used.

CHAPTER 5

Using GRT

5.1 From Ada

Warning: This topic is only for advanced users who know how to use *Ada* and *GNAT*. This is provided only for reference; we have tested this once before releasing *GHDL* 0.19, but this is not checked at each release.

The simulator kernel of *GHDL* named *GRT* is written in *Ada95* and contains a very light and slightly adapted version of *VHPI*. Since it is an *Ada* implementation it is called *AVHPI*. Although being tough, you may interface to *AVHPI*.

For using *AVHPI*, you need the sources of *GHDL* and to recompile them (at least the *GRT* library). This library is usually compiled with a *No_Run_Time* pragma, so that the user does not need to install the *GNAT* runtime library. However, you certainly want to use the usual runtime library and want to avoid this pragma. For this, reset the *GRT_PRAGMA_FLAG* variable.

```
$ make GRT_PRAGMA_FLAG= grt-all
```

Since *GRT* is a self-contained library, you don't want *gnatlink* to fetch individual object files (furthermore this doesn't always work due to tricks used in *GRT*). For this, remove all the object files and make the `.ali` files read-only.

```
$ rm *.o
$ chmod -w *.ali
```

You may then install the sources files and the `.ali` files. I have never tested this step.

You are now ready to use it.

Here is an example, `test_grt.adb` which displays the top level design name.

```
with System; use System;
with Grt.Avhpi; use Grt.Avhpi;
with Ada.Text_IO; use Ada.Text_IO;
with Ghdl_Main;

procedure Test_Grt is
  --  VHPI handle.
```

(continues on next page)

(continued from previous page)

```
  H : VhpiHandleT;
  Status : Integer;

  --  Name.
  Name : String (1 .. 64);
  Name_Len : Integer;
begin
  --  Elaborate and run the design.
  Status := Ghdl_Main (0, Null_Address);

  --  Display the status of the simulation.
  Put_Line ("Status is " & Integer'Image (Status));

  --  Get the root instance.
  Get_Root_Inst(H);

  --  Disp its name using vhpi API.
  Vhpi_Get_Str (VhpiNameP, H, Name, Name_Len);
  Put_Line ("Root instance name: " & Name (1 .. Name_Len));
end Test_Grt;
```

First, analyze and bind your design:

```
$ ghdl -a counter.vhdl
$ ghdl --bind counter
```

Then build the whole:

```
$ gnatmake test_grt -aL`grt_ali_path` -aI`grt_src_path` -largs
 `ghdl --list-link counter`
```

Finally, run your design:

```
$ ./test_grt
Status is  0
Root instance name: counter
```

CHAPTER 6

Examples

6.1 Quick Start

6.1.1 random

By default, GHDL includes the standard C library in the generated simulation models. Hence, resources from `stdlib` can be used without any modification to the build procedure.

This example shows how to import and use `rand` to generate and print 10 integer numbers. The VHDL code is equivalent to the following C snippet. However, note that this C source is NOT required, because `stdlib` is already built in.

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

int main (void) {
  int i;
  for (i = 0; i < 10; i++)
    printf ("%d\n", rand ());
  return 0;
}
```

6.1.2 math

By the same token, it is possible to include functions from system library by just providing the corresponding linker flag.

In this example, function `sin` from the `math` library is used to compute 10 values. As in the previous example, no additional C sources are required, because the `math` library is already compiled and installed in the system.

6.1.3 customc

When the required functionality is not available in pre-built libraries, custom C sources and/or objects can be added to the elaboration and/or linking.

This example shows how to bind custom C functions in VHDL as either procedures or functions. Four cases are included: `custom_procedure`, `custom_procedure_withargs`, `custom_function` and `custom_function_withargs`. In all cases, the parameters are defined as integers, in order to keep it simple. See *Type declarations* for further details.

Since either C sources or pre-compiled `.o` objects can be added, in C/C++ projects of moderate complexity, it might be desirable to merge all the C sources in a single object before elaborating the design.

6.1.4 package

If the auxillary VHPIDIRECT subprograms need to be accessed in more than one entity, it is possible to package the subprograms. This also makes it very easy to reuse the VHPIDIRECT declarations in different projects.

In this example two different entities use a C defined `c_printInt(val: integer)` subprogram to print two different numbers. Subprogram declaration requirements are detailed under the *Type declarations* section.

6.1.5 sharedvar

While sharing variables through packages in VHDL 1993 is flexible, in VHDL 2008 protected types need to be used. However, GHDL allows to relax some rules of the LRM through `-frelaxed`.

This example shows multiple alternatives to share variables through packages, depending on the target version of the standard. Three different binaries are built from the same entity, using:

- A VHDL 1993 package with `--std=93`.
- A VHDL 1993 package with `--std=08 -frelaxed`.
- A VHDL 2008 package with `--std=08`.

Note: Procedure `setVar` is not strictly required. It is used to allow the same descriptions of the entity/architectures to work with both VHDL 1993 and VHDL 2008. See the bodies of the procedure in pkg_93.vhd and pkg_08.vhd.

As an alternative to using a shared variable in VHDL, subdir shint contains an approach based on a helper record type which is used as a handle. Mimicking the concept of *methods* from Object Oriented (OO) programming, helper C functions are used to read/write the actual variables, instead of sharing data through an access/pointer. This approach is more verbose than others, but it works with either VHDL 1993 or VHDL 2008 without modification and without requiring `-frelaxed`. Moreover, it enhances encapsulation, as it provides a user-defined API between VHDL and C, which can improve maintainability when sources are reused. As a matter of fact, this approach is found in verification projects such as VUnit and OSVVM.

6.2 Wrapping

6.2.1 basic

Instead of using GHDL's own entrypoint to the execution, it is possible to wrap it by providing a custom `main` function. Upon existence of `main`, execution of the simulation is triggered by calling `ghdl_main`.

This is the most basic example of such usage. `ghdl_main` is declared as `extern` in C, and arguments `argc` and `argv` are passed without modification. However, this sets the ground for custom prepocessing and postprocessing in a foreign language.

Other options are to just pass empty arguments (`ghdl_main(0, NULL)`) or to customize them:

```c
char* args[] = {NULL, "--wave=wave.ghw"};
ghdl_main(2, args);
```

See *Wrapping a simulation (ghdl_main)* for further details about the constraints of `argv`.

6.2.2 time

Although most of the provided examples are written in C, VHPIDIRECT can be used with any language that supports a C-alike compile and link model.

This example shows how to time the execution of a simulation from either C or Ada. In both cases, function `clock` is used to get the time before and after calling `ghdl_main`. Regarding the build procedure, it is to be noted that C sources are elaborated with `-e`, because GHDL allows to pass parameters (in this case, additional C sources) to the compiler and/or linker. However, since it is not possible to do so with Ada, `gnatmake`, `--bind` and `--list-link` are used instead. See *Linking object files* for further info about custom linking setups.

Hint: Compared to the previous example, the declaration of `ghdl_main` includes three arguments in this example: `int argc, void** argv, void** envp`. This is done for illustration purposes only, as it has no real effect on the exercise.

6.3 Linking

6.3.1 bind

Although GHDL's elaborate command can compile and link C sources, it is sometimes preferred or required to call a compiler explicitly with custom arguments. This is useful, e.g., when a simulation is to be embedded in the build of an existing C/C++ application.

This example is equivalent to *basic*, but it shows how to use `--bind` and `--list-link` instead of `-e`. See *Linking object files* for further details.

Hint: Objects generated by `--bind` are created in the working directory. See GCC/LLVM only commands and ghdl#781.

6.4 Shared

Important: As explained in *Loading a simulation*, in order to load binaries/libraries dynamically, those need to be built as position independent code/executables (PIC/PIE).

6.4.1 shlib

This example features the same functionality as *random*. However, custom C sources are use (as in *customc*) and these are built as a shared library. See *Loading foreign objects from within a simulation* for further info.

6.4.2 dlopen

Although this example does not include a simulation built with GHDL, it is a test and the introduction to the next example. In this test, two separate shared libraries are built from C sources, both including a function named `ghdl_main`. Then, in a main C application, both shared libraries are dynamically loaded at the same time, and both are executed (one after the other)

This example tests whether symbol `ghdl_main` is visible in the shared libraries, and whether the same symbol name can be loaded from multiple shared libraries (and used) at the same time.

Tip: If the symbol is not found, try adding *-g*, *-rdynamic* and/or *-O0* when building the shared libraries. Tools such as `objdump`, `readelf` or `nm` can be used to check if a symbol is visible. For instance, `objdump -d corea.so | grep ghdl_main`.

Hint: Building multiple designs as separate artifacts and dynamically loading them at the same time is a naive approach to multi-core simulation with GHDL. It is also a possible solution for coarse grained co-simulation with Verilator.

6.4.3 shghdl

This example is complementary to *shlib*, since the VHDL simulation is built as a shared library, which is then loaded from a main C application (as in *dlopen*).

When `main` is executed:

- The shared libray is loaded, symbol `print_something` is searched for, and it is executed.
- Symbol `ghdl_main` is searched for, and it is executed three times. Unfortunately, GHDL does not currently support reseting/restarting the simulation runtime. Hence, in this example the shared library is unloaded and loaded again before calling `ghdl_main` after the first time.

See Generating_shared_libraries for further details with regard to the visibility of symbols in the shared libraries.

Note: On GNU/linux, both executable binaries and shared libraries use the ELF format. As a result, although hackish, it is possible to load an executable binary dynamically, i.e. without using any of the `shared` options explained in Generating_shared_libraries. In this example, this case is also tested. However, this is not suggested at all, since it won't work on all platforms.

6.5 Arrays

6.5.1 logicvector

Commonly signals in VHDL are of a logic type or a vector thereof (`std_logic` and `std_logic_vector`), coming from IEEE's `std_logic_1164` package. These types can hold values other than high and low (`1` and `0`) and are enumerated as:

0. 'U'
1. 'X'
2. '0'
3. '1'
4. 'Z'
5. 'W'
6. 'L'
7. 'H'
8. '-'

As mentioned in *Restrictions on type declarations*:

- Because the number of enumeration values is less than 256, logic values are transported in 8 bit words (a `char` type in C).
  - In this example two declarations make handling logic values in C a bit easier:
    - Providing logic values in C as their enumeration numbers is simplified with the use of a matching enumeration, `HDL_LOGIC_STATES`.
    - Printing out a logic value's associated character is also simplified with the `const char HDL_LOGIC_CHAR[]` declaration.
- Logic vectors, of a bounded size, can be easily created in C as a `char[]` and passed to VHDL to be read as an `access` type in VHDL, in this case an access of a subtype of std_logic_vector.

This example builds on the integer vector example (COSIM:VHPIDIRECT:Examples:arrays:intvector), by instead passing an array of logic values. Foreign subprograms are declared that enable receiving the size of two different logic vectors as well as the vectors themselves from C. There is only one subprogram to get the size of both C arrays, and it takes in an integer to determine which array's size gets returned.

Hint: The `getLogicVecSize` in VHDL is declared as receiving a `boolean` argument. In C the function is declared to receive an `char` argument. The VHDL booleans `false` and `true` are enumerations, and have integer values, `0` and `1` respectively. As with the logic values, the boolean enumerations use fewer than 8 bits, so the single byte in C's `char` variable receives the VHDL enumeration correctly.

For illustrative purposes, the two vectors are populated with logic values in different ways:

- LogicVectorA's indices are manually filled with enumeration values from HDL_LOGIC_STATES.

  -
    ```
    logic_vec_A[0] = HDL_U;
    ```
- LogicVectorB's indices are filled with an integer value.

  -
    ```
    for(int i = 0; i < SIZE_LOGIC_VEC_B; i++){
      logic_vec_B[i] = 8-i;
    }
    ```

Attention: The integer values that are given to `char` variables in C which are intended to be read as VHDL logic values, must be limited to [0, 8]. This ensures that they represent one of the 9 enumerated logic values.

6.6 Other co-simulation projects

This sections contains references to other co-simulation projects based on GHDL and VHPIDIRECT.

6.6.1 VUnit

VUnit is an open source unit testing framework for VHDL/SystemVerilog. Sharing memory buffers between foreign C or Python applications and VHDL testbenches is supported through GHDL's VHPIDIRECT features. Buffers are accessed from VHDL as either strings, arrays of bytes or arrays of 32 bit integers. See VUnit example external buffer for details about the API.

6.6.2 ghdlex and netpp

netpp (network property protocol) is a communication library allowing to expose variables or other properties of an application to the network as abstract 'Properties'. Its basic philosophy is that a device always knows its capabilities. netpp capable devices can be explored by command line, Python scripts or GUI applications.

Properties of a device - be it virtual or real - are typically described by a static description in an XML device description language, but they can also be constructed on the fly.

ghdlex is a set of C extensions to facilitate data exchange between a GHDL simulation and external applications. VHPIDIRECT mechanisms are used to wrap GHDL data types into structures usable from a C library. *ghdlex* uses the netpp library to expose virtual entities (such as pins or RAM) to the network. It also demonstrates simple data I/O through unix pipes. A few VHDL example entities are provided, such as a virtual console, FIFOs, RAM.

The author of *netpp* and *ghdlex* is also working on MaSoCist, a linux'ish build system for System on Chip designs, based on GHDL. It allows to handle more complex setup, e.g. how a RISC-V architecture (for example) is regress-tested using a virtual debug interface.

Interfacing with foreign languages through VHPIDIRECT is possible on any platform. You can define a subprogram in a foreign language (such as *C* or *Ada*) and import it into a VHDL design.

Note: GHDL supports different backends, and not all of them generate binary artifacts. Precisely, `mcode` is an in-memory backend. Hence, the examples need to be built/executed with either LLVM or GCC backends. A few of them, the ones that do not require linking object files, can be used with mcode.

Attention: As a consequence of the runtime copyright, you are not allowed to distribute an executable produced by GHDL without allowing access to the VHDL sources. See Copyrights | Licenses.

Tip: See ghdl#1053 for on-going work with regard to VHPIDIRECT.

CHAPTER 7

Examples

TBC

See VPI build commands.

TBC

Index