GHDL Cosimulation Documentation

Release latest

Tristan Gingold and contributors

Apr 18, 2020

VHPIDIRECT

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/C++ models à la Verilator is not supported. However, a *ghdlator* might be available in the future.

CHAPTER 1

Type declarations

Only subprograms (functions or procedures) can be imported, using the foreign attribute. In this 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;
```

A subprogram is made foreign if the *foreign* attribute decorates it. This attribute is declared in the 1993 revision of the `std.standard` package. Therefore, you cannot use this feature in VHDL 1987.

The decoration is achieved through an attribute specification. 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.

Even when a subprogram is foreign, its body must be present. However, since it won't be called, you can make it empty or simply put an assertion.

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 object file with the source code for the foreign subprogram must then be linked to GHDL, expanded upon in *Linking object files*.

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 They are represented by a 32 bit word. This generally corresponds to *int* for *C* or *Integer* for *Ada*.

physical types They are represented by a 64 bit word. This generally corresponds to the *long long* for *C* or *Long_Long_Integer* for *Ada*.

floating point types They are represented by a 64 bit floating point word. This generally corresponds to *double* for *C* or *Long_Float* for *Ada*.

enumeration types They are represented by an 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. The *out* and *inout* interfaces of non-composite types are gathered in a record and this record is passed by reference as the first argument to the subprogram. As a consequence, you shouldn't use *out* and *inout* modes in foreign subprograms, since they are not portable.

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. Do not use unconstrained arrays in foreign subprograms.

Accesses to an unconstrained array are fat pointers. Other accesses correspond to an address 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 compiled to an object file. Analysis must be made of the HDL files, then elaboration with `-e -Wl,personal.o [options...] primary_unit [secondary_unit]` as arguments. Additional 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 Loading a simulation

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.). For example, in 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
```

(continues on next page)

(continued from previous page)

```
_ctypes.dlclose(gbin._handle)
# On Windows
#_ctypes.FreeLibrary(gbin._handle)
```

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.

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

Attention: By default, GHDL uses `grt.ver` to limit which symbols are exposed in the generated binary, and `ghdl_main` is not included. Hence, the version script needs to be removed, or a complementary script needs to be provided. Otherwise, it will not be possible to find the function easily. See `--list-link` for further info.

Tip: See #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

Important: This sections contains advanced examples using specific features of the language, the tool, or interaction with third-party projects. It is suggested for users who are new to either *GHDL* or *VHDL* to read Quick Start Guide first.

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.

6.2 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:

```
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 #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 `ghdl_main` is searched for, and it is executed. Unfortunately, GHL does not make `ghdl_main` visible by default. Hence, if a simulation model is to be loaded dynamically, visibility needs to be tweaked. This is also true for any additional function that is described in the C sources that are linked to the simulation model.

- It is possible to force a symbol to be added with `-Wl,-Wl,-u,ghdl_main`.
- If the shared library is built with `-e`, option `-Wl,-Wl,--version-script=file.ver` can be used, where `file.ver` is an additional custom version file such as:

```
VHPIDIRECT {
  global:
    ghdl_main;
  local:
    *;
};
```

- [**EXPERIMENTAL** #1184] Alternatively, `-shared` removes the version script.
- 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 #640), and make all symbols visible by default. It is also possible to pass an additional script. See description of `--list-link` for further details.

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:

6.5 Other co-simulation projects

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

6.5.1 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.

6.5.2 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 #1053 for on-going work with regard to VHPIDIRECT.

CHAPTER 7

Examples

Important: This sections contains advanced examples using specific features of the language, the tool, or interaction with third-party projects. It is suggested for users who are new to either *GHDL* or *VHDL* to read Quick Start Guide first.

TBC

See VPI build commands.

TBC

Index