GHDL Documentation

Release 2017-03-01

Tristan Gingold

Sep 14, 2017

Contents

Contents:

CHAPTER 1

Introduction

Content of this manual

This manual is the user and reference manual for GHDL. It does not contain an introduction to VHDL. Thus, the reader should have at least a basic knowledge of VHDL. A good knowledge of VHDL language reference manual (usually called LRM) is a plus.

What is *VHDL*?

VHDL is an acronym for Very High Speed Integrated Circuit Hardware Description Language which is a programming language used to describe a logic circuit by function, data flow behaviour, or structure.

VHDL is a programming language: although *VHDL* was not designed for writing general purpose programs, you can write any algorithm with the *VHDL* language. If you are able to write programs, you will find in *VHDL* features similar to those found in procedural languages such as *C*, *Python*, or *Ada*. *VHDL* derives most of its syntax and semantics from *Ada*. Knowing *Ada* is an advantage for learning *VHDL* (it is an advantage in general as well).

However, *VHDL* was not designed as a general purpose language but as an *HDL* (hardware description language). As the name implies, *VHDL* aims at modeling or documenting electronics systems. Due to the nature of hardware components which are always running, *VHDL* is a highly concurrent language, built upon an event-based timing model.

Like a program written in any other language, a *VHDL* program can be executed. Since *VHDL* is used to model designs, the term *simulation* is often used instead of *execution*, with the same meaning.

Like a program written in another hardware description language, a *VHDL* program can be transformed with a *synthesis tool* into a netlist, that is, a detailed gate-level implementation.

What is *GHDL*?

GHDL is a shorthand for G Hardware Design Language. Currently, *G* has no meaning.

GHDL is a *VHDL* compiler that can execute (nearly) any *VHDL* program. *GHDL* is *not* a synthesis tool: you cannot create a netlist with *GHDL*.

Unlike some other simulators, *GHDL* is a compiler: it directly translates a *VHDL* file to machine code, using the *GCC* or *LLVM* back-end and without using an intermediary language such as *C* or *C++*. Therefore, the compiled code should be faster and the analysis time should be shorter than with a compiler using an intermediary language.

The Windows(TM) version of *GHDL* is not based on *GCC* but on an internal code generator.

The current version of *GHDL* does not contain any graphical viewer: you cannot see signal waves. You can still check with a test bench. The current version can produce a *VCD* file which can be viewed with a wave viewer, as well as *ghw* files to be viewed by *gtkwave*.

GHDL aims at implementing *VHDL* as defined by IEEE 1076. It supports most of the 1987 standard and most features added by the 1993 standard.

CHAPTER 2

Starting with GHDL

In this chapter, you will learn how to use the GHDL compiler by working on two examples.

The hello world program

To illustrate the large purpose of VHDL, here is a commented VHDL "Hello world" program.

```
--  Hello world program.
use std.textio.all; -- Imports the standard textio package.

--  Defines a design entity, without any ports.
entity hello_world is
end hello_world;

architecture behaviour of hello_world is
begin
   process
      variable l : line;
   begin
      write (l, String'("Hello world!"));
      writeline (output, l);
      wait;
   end process;
end behaviour;
```

Suppose this program is contained in the file `hello.vhdl`. First, you have to compile the file; this is called *analysis* of a design file in VHDL terms.

```
$ ghdl -a hello.vhdl
```

This command creates or updates a file `work-obj93.cf`, which describes the library *work*. On GNU/Linux, this command generates a file `hello.o`, which is the object file corresponding to your VHDL program. The object file is not created on Windows.

Then, you have to build an executable file.

```
$ ghdl -e hello_world
```

The `-e` option means *elaborate*. With this option, *GHDL* creates code in order to elaborate a design, with the `hello_world` entity at the top of the hierarchy.

On GNU/Linux, if you have enabled the GCC backend during the compilation of *GHDL*, an executable program called `hello_world` which can be run is generated:

```
$ ghdl -r hello_world
```

or directly:

```
$ ./hello_world
```

On Windows or if the GCC backend was not enabled, no file is created. The simulation is launched using this command:

```
> ghdl -r hello_world
```

The result of the simulation appears on the screen:

```
Hello world!
```

A full adder

VHDL is generally used for hardware design. This example starts with a full adder described in the `adder.vhdl` file:

```
entity adder is
  -- `i0`, `i1` and the carry-in `ci` are inputs of the adder.
  -- `s` is the sum output, `co` is the carry-out.
  port (i0, i1 : in bit; ci : in bit; s : out bit; co : out bit);
end adder;

architecture rtl of adder is
begin
   --  This full-adder architecture contains two concurrent assignment.
   --  Compute the sum.
   s <= i0 xor i1 xor ci;
   --  Compute the carry.
   co <= (i0 and i1) or (i0 and ci) or (i1 and ci);
end rtl;
```

You can analyze this design file:

```
$ ghdl -a adder.vhdl
```

You can try to execute the *adder* design, but this is useless, since nothing externally visible will happen. In order to check this full adder, a testbench has to be run. This testbench is very simple, since the adder is also simple: it checks exhaustively all inputs. Note that only the behaviour is tested, timing constraints are not checked. The file `adder_tb.vhdl` contains the testbench for the adder:

```
--  A testbench has no ports.
entity adder_tb is
end adder_tb;

architecture behav of adder_tb is
   --  Declaration of the component that will be instantiated.
   component adder
     port (i0, i1 : in bit; ci : in bit; s : out bit; co : out bit);
   end component;

   --  Specifies which entity is bound with the component.
   for adder_0: adder use entity work.adder;
   signal i0, i1, ci, s, co : bit;
begin
   --  Component instantiation.
   adder_0: adder port map (i0 => i0, i1 => i1, ci => ci,
                            s => s, co => co);

   --  This process does the real job.
   process
      type pattern_type is record
         --  The inputs of the adder.
         i0, i1, ci : bit;
         --  The expected outputs of the adder.
         s, co : bit;
      end record;
      --  The patterns to apply.
      type pattern_array is array (natural range <>) of pattern_type;
      constant patterns : pattern_array :=
        (('0', '0', '0', '0', '0'),
         ('0', '0', '1', '1', '0'),
         ('0', '1', '0', '1', '0'),
         ('0', '1', '1', '0', '1'),
         ('1', '0', '0', '1', '0'),
         ('1', '0', '1', '0', '1'),
         ('1', '1', '0', '0', '1'),
         ('1', '1', '1', '1', '1'));
   begin
      --  Check each pattern.
      for i in patterns'range loop
         --  Set the inputs.
         i0 <= patterns(i).i0;
         i1 <= patterns(i).i1;
         ci <= patterns(i).ci;
         --  Wait for the results.
         wait for 1 ns;
         --  Check the outputs.
         assert s = patterns(i).s
            report "bad sum value" severity error;
         assert co = patterns(i).co
            report "bad carry out value" severity error;
      end loop;
      assert false report "end of test" severity note;
      --  Wait forever; this will finish the simulation.
      wait;
   end process;
end behav;
```

As usual, you should analyze the design:

```
$ ghdl -a adder_tb.vhdl
```

And build an executable for the testbench:

```
$ ghdl -e adder_tb
```

You do not need to specify which object files are required: GHDL knows them and automatically adds them in the executable. Now, it is time to run the testbench:

```
$ ghdl -r adder_tb
adder_tb.vhdl:52:7:(assertion note): end of test
```

If your design is rather complex, you'd like to inspect signals. Signals value can be dumped using the VCD file format. The resulting file can be read with a wave viewer such as GTKWave. First, you should simulate your design and dump a waveform file:

```
$ ghdl -r adder_tb --vcd=adder.vcd
```

Then, you may now view the waves:

```
$ gtkwave adder.vcd
```

See *Simulation options*, for more details on the `--vcd` option and other runtime options.

Starting with a design

Unless you are only studying VHDL, you will work with bigger designs than the ones of the previous examples.

Let's see how to analyze and run a bigger design, such as the DLX model suite written by Peter Ashenden which is distributed under the terms of the GNU General Public License. A copy is kept on http://ghdl.free.fr/dlx.tar.gz

First, untar the sources:

```
$ tar zxvf dlx.tar.gz
```

In order not to pollute the sources with the library, it is a good idea to create a `work/` subdirectory for the *WORK* library. To any GHDL commands, we will add the `--workdir=work` option, so that all files generated by the compiler (except the executable) will be placed in this directory.

```
$ cd dlx
$ mkdir work
```

We will run the `dlx_test_behaviour` design. We need to analyze all the design units for the design hierarchy, in the correct order. GHDL provides an easy way to do this, by importing the sources:

```
$ ghdl -i --workdir=work *.vhdl
```

and making a design:

```
$ ghdl -m --workdir=work dlx_test_behaviour
```

Before this second stage, GHDL knows all the design units of the DLX, but no one have been analyzed. The make command of GHDL analyzes and elaborates a design. This creates many files in the `work/` directory, and the `dlx_test_behaviour` executable in the current directory.

The simulation needs to have a DLX program contained in the file `dlx.out`. This memory image will be be loaded in the DLX memory. Just take one sample:

```
$ cp test_loop.out dlx.out
```

And you can run the test suite:

```
$ ghdl -r --workdir=work dlx_test_behaviour
```

The test bench monitors the bus and displays each instruction executed. It finishes with an assertion of severity level note:

```
dlx-behaviour.vhdl:395:11:(assertion note): TRAP instruction
 encountered, execution halted
```

Since the clock is still running, you have to manually stop the program with the `C-c` key sequence. This behavior prevents you from running the test bench in batch mode. However, you may force the simulator to stop when an assertion above or equal a certain severity level occurs:

```
$ ghdl -r --workdir=work dlx_test_behaviour --assert-level=note
```

With this option, the program stops just after the previous message:

```
dlx-behaviour.vhdl:395:11:(assertion note): TRAP instruction
 encountered, execution halted
error: assertion failed
```

If you want to make room on your hard drive, you can either:

- clean the design library with the GHDL command:

  ```
  $ ghdl --clean --workdir=work
  ```

  This removes the executable and all the object files. If you want to rebuild the design at this point, just do the make command as shown above.

- remove the design library with the GHDL command:

  ```
  $ ghdl --remove --workdir=work
  ```

  This removes the executable, all the object files and the library file. If you want to rebuild the design, you have to import the sources again, and to make the design.

- remove the `work/` directory:

  ```
  $ rm -rf work
  ```

  Only the executable is kept. If you want to rebuild the design, create the `work/` directory, import the sources, and make the design.

Sometimes, a design does not fully follow the VHDL standards. For example it uses the badly engineered `std_logic_unsigned` package. GHDL supports this VHDL dialect through some options:

```
--ieee=synopsys -fexplicit
```

See *IEEE library pitfalls*, for more details.

CHAPTER 3

Invoking GHDL

The form of the **ghdl** command is:

```
ghdl command [options...]
```

The GHDL program has several commands. The first argument selects the command. The options are used to slightly modify the action.

No option is allowed before the command. Except for the run command, no option is allowed after a filename or a unit name.

If the number of options is large and the command line length is beyond the system limit, you can use a response file. An argument that starts with a `@` is considered as a response file; it is replaced by arguments read from the file (separated by blanks and end of line).

Building commands

The mostly used commands of GHDL are those to analyze and elaborate a design.

Analysis command

Analyze one or several files:

```
ghdl -a [options...] file...
```

The analysis command compiles one or more files, and creates an object file for each source file. The analysis command is selected with `-a` switch. Any argument starting with a dash is an option, the others are filenames. No options are allowed after a filename argument. GHDL analyzes each filename in the given order, and stops the analysis in case of error (the following files are not analyzed).

See *GHDL options*, for details on the GHDL options. For example, to produce debugging information such as line numbers, use:

```
ghdl -a -g my_design.vhdl
```

Elaboration command

Elaborate a design:

```
ghdl -e [options..] primary_unit [secondary_unit]
```

On GNU/Linux, if the GCC backend was enabled during the compilation of *GHDL*, the elaboration command creates an executable containing the code of the *VHDL* sources, the elaboration code and simulation code to execute a design hierarchy. The executable is created in the current directory. On Windows or if the GCC backend was not enabled, this command elaborates the design but does not generate anything.

The elaboration command is selected with `-e` switch, and must be followed by either:

- a name of a configuration unit
- a name of an entity unit
- a name of an entity unit followed by a name of an architecture unit

Name of the units must be a simple name, without any dot. You can select the name of the *WORK* library with the `--work=NAME` option, as described in *GHDL options*.

See *Top entity*, for the restrictions on the root design of a hierarchy.

On GNU/Linux the filename of the executable is the name of the primary unit, or for the later case, the concatenation of the name of the primary unit, a dash, and the name of the secondary unit (or architecture). On Windows there is no executable generated.

The `-o` followed by a filename can override the default executable filename.

For the elaboration command, *GHDL* re-analyzes all the configurations, entities, architectures and package declarations, and creates the default configurations and the default binding indications according to the LRM rules. It also generates the list of objects files required for the executable. Then, it links all these files with the runtime library.

The actual elaboration is performed at runtime.

On Windows this command can be skipped because it is also done by the run command.

Run command

Run (or simulate) a design:

```
ghdl -r [options...] primary_unit [secondary_unit] [simulation_options...]
```

The options and arguments are the same as for the elaboration command, *Elaboration command*.

On GNU/Linux this command simply determines the filename of the executable and executes it. Options are ignored. You may also directly execute the program. The executable must be in the current directory.

This command exists for three reasons:

- You don't have to create the executable program name.
- It is coherent with the `-a` and `-e` commands.
- It works with the Windows implementation, where the code is generated in memory.

On Windows this command elaborates and launches the simulation. As a consequence you must use the same options used during analysis.

See *Simulation and runtime*, for details on options.

Elaborate and run command

Elaborate and then simulate a design unit:

```
ghdl --elab-run [elab_options...] primary_unit [secondary_unit] [run_options...]
```

This command acts like the elaboration command (see *Elaboration command*) followed by the run command (see *Run command*).

Bind command

Bind a design unit and prepare the link step:

```
ghdl --bind [options] primary_unit [secondary_unit]
```

This command is only available on GNU/Linux.

This performs only the first stage of the elaboration command; the list of objects files is created but the executable is not built. This command should be used only when the main entry point is not ghdl.

Link command

Link an already bound design unit:

```
ghdl --link [options] primary_unit [secondary_unit]
```

This performs only the second stage of the elaboration command: the executable is created by linking the files of the object files list. This command is available only for completeness. The elaboration command is equivalent to the bind command followed by the link command.

List link command

Display files which will be linked:

```
ghdl --list-link primary_unit [secondary_unit]
```

This command is only available on GNU/Linux.

This command may be used only after a bind command. GHDL displays all the files which will be linked to create an executable. This command is intended to add object files in a link of a foreign program.

Check syntax command

Analyze files but do not generate code:

```
ghdl -s [options] files
```

This command may be used to check the syntax of files. It does not update the library.

Analyze and elaborate command

Analyze files and elaborate them at the same time.

On GNU/Linux:

```
ghdl -c [options] file... -e primary_unit [secondary_unit]
```

On Windows:

```
ghdl -c [options] file... -r primary_unit [secondary_unit]
```

This command combines analysis and elaboration: files are analyzed and the unit is then elaborated. However, code is only generated during the elaboration. On Windows the simulation is launched.

To be more precise, the files are first parsed, and then the elaboration drives the analysis. Therefore, there is no analysis order, and you don't need to care about it.

All the units of the files are put into the *work* library. But, the work library is neither read from disk nor saved. Therefore, you must give all the files of the *work* library your design needs.

The advantages over the traditional approach (analyze and then elaborate) are:

- The compilation cycle is achieved in one command.
- Since the files are only parsed once, the compilation cycle may be faster.
- You don't need to know an analysis order
- This command produces smaller executable, since unused units and subprograms do not generate code.

However, you should know that currently most of the time is spent in code generation and the analyze and elaborate command generate code for all units needed, even units of `std` and `ieee` libraries. Therefore, according to the design, the time for this command may be higher than the time for the analyze command followed by the elaborate command.

This command is still experimental. In case of problems, you should go back to the traditional way.

GHDL options

Besides the options described below, *GHDL* passes any debugging options (those that begin with `-g`) and optimizations options (those that begin with `-O` or `-f`) to *GCC*. Refer to the *GCC* manual for details.

--workdir=<DIR>
: Specify the directory where the `WORK` library is located. When this option is not present, the `WORK` library is in the current directory. The object files created by the compiler are always placed in the same directory as the `WORK` library.

  Use option *-P* to specify where libraries other than `WORK` are placed.

--std=<STD>
: Specify the standard to use. By default, the standard is `93c`, which means VHDL-93 accepting VHDL-87 syntax. For details on `STD` values see *VHDL standards*.

--ieee=<VER>
: Select the `IEEE` library to use. `VER` must be one of:

  none Do not supply an *IEEE* library. Any library clause with the `IEEE` identifier will fail, unless you have created by your own a library with the *IEEE* name.

standard Supply an *IEEE* library containing only packages defined by `ieee` standards. Currently, there are the multivalue logic system packages `std_logic_1164` defined by IEEE 1164, the synthesis packages , `numeric_bit` and `numeric_std` defined by IEEE 1076.3, and the `vital` packages `vital_timing` and `vital_primitives`, defined by IEEE 1076.4. The version of these packages is defined by the VHDL standard used. See *VITAL packages*, for more details.

synopsys Supply the former packages and the following additional packages: `std_logic_arith`, `std_logic_signed`, `std_logic_unsigned`, `std_logic_textio`.

These packages were created by some companies, and are popular. However they are not standard packages, and have been placed in the *IEEE* library without the permission from the `ieee`.

mentor Supply the standard packages and the following additional package: `std_logic_arith`. The package is a slight variation of a definitely not standard but widely mis-used package.

To avoid errors, you must use the same *IEEE* library for all units of your design, and during elaboration.

-P<DIRECTORY>
Add *DIRECTORY* to the end of the list of directories to be searched for library files. A library is searched in *DIRECTORY* and also in *DIRECTORY/LIB/vVV* (where *LIB* is the name of the library and *VV* the vhdl standard).

The *WORK* library is always searched in the path specified by the `--workdir=` option, or in the current directory if the latter option is not specified.

-fexplicit
When two operators are overloaded, give preference to the explicit declaration. This may be used to avoid the most common pitfall of the `std_logic_arith` package. See *IEEE library pitfalls*, for an example.

This option is not set by default. I don't think this option is a good feature, because it breaks the encapsulation rule. When set, an operator can be silently overridden in another package. You'd better to fix your design and use the `numeric_std` package.

-frelaxed-rules
Within an object declaration, allow to reference the name (which references the hidden declaration). This ignores the error in the following code:

```
package pkg1 is
 type state is (state1, state2, state3);
end pkg1;

use work.pkg1.all;
package pkg2 is
 constant state1 : state := state1;
end pkg2;
```

Some code (such as Xilinx packages) have such constructs, which are valid.

(The scope of the `state1` constant start at the *constant* word. Because the constant `state1` and the enumeration literal `state1` are homograph, the enumeration literal is hidden in the immediate scope of the constant).

This option also relaxes the rules about pure functions. Violations result in warnings instead of errors.

-fpsl
Enable parsing of PSL assertions within comments. See *PSL implementation*, for more details.

--no-vital-checks

--vital-checks
Disable or enable checks of restriction on VITAL units. Checks are enabled by default.

Checks are performed only when a design unit is decorated by a VITAL attribute. The VITAL attributes are `VITAL_Level0` and `VITAL_Level1`, both declared in the `ieee.VITAL_Timing` package.

Currently, VITAL checks are only partially implemented. See *VHDL restrictions for VITAL*, for more details.

--syn-binding
Use synthesizer rules for component binding. During elaboration, if a component is not bound to an entity using VHDL LRM rules, try to find in any known library an entity whose name is the same as the component name.

This rule is known as synthesizer rule.

There are two key points: normal VHDL LRM rules are tried first and entities are searched only in known library. A known library is a library which has been named in your design.

This option is only useful during elaboration.

--PREFIX=<PATH>
Use `PATH` as the prefix path to find commands and pre-installed (std and ieee) libraries.

--GHDL1=<COMMAND>
Use `COMMAND` as the command name for the compiler. If `COMMAND` is not a path, then it is searched in the path.

--AS=<COMMAND>
Use `COMMAND` as the command name for the assembler. If `COMMAND` is not a path, then it is searched in the path. The default is `as`.

--LINK=<COMMAND>
Use `COMMAND` as the linker driver. If `COMMAND` is not a path, then it is searched in the path. The default is `gcc`.

-v
Be verbose. For example, for analysis, elaboration and make commands, GHDL displays the commands executed.

Passing options to other programs

These options are only available on GNU/Linux.

For many commands, *GHDL* acts as a driver: it invokes programs to perform the command. You can pass arbitrary options to these programs.

Both the compiler and the linker are in fact GCC programs. See the GCC manual for details on GCC options.

-Wc,<OPTION>
Pass *OPTION* as an option to the compiler.

-Wa,<OPTION>
Pass *OPTION* as an option to the assembler.

-Wl,<OPTION>
Pass *OPTION* as an option to the linker.

GHDL Diagnostics Control

-fcolor-diagnostics

-fno-color-diagnostics
Control whether diagnostic messages are displayed in color. The default is on when the standard output is a terminal.

-fdiagnostics-show-option

-fno-diagnostics-show-option
Control whether the warning option is displayed at the end of warning messages, so that user can easily know how to disable it.

GHDL warnings

Some constructions are not erroneous but dubious. Warnings are diagnostic messages that report such constructions. Some warnings are reported only during analysis, others during elaboration.

You could disable a warning by using the `--warn-no-XXX` or `-Wno-XX` instead of `--warn-XXX` or `-WXXX`.

--warn-reserved
Emit a warning if an identifier is a reserved word in a later VHDL standard.

--warn-default-binding
During analyze, warns if a component instantiation has neither configuration specification nor default binding. This may be useful if you want to detect during analyze possibly unbound component if you don't use configuration. *VHDL standards*, for more details about default binding rules.

--warn-binding
During elaboration, warns if a component instantiation is not bound (and not explicitly left unbound). Also warns if a port of an entity is not bound in a configuration specification or in a component configuration. This warning is enabled by default, since default binding rules are somewhat complex and an unbound component is most often unexpected.

However, warnings are even emitted if a component instantiation is inside a generate statement. As a consequence, if you use the conditional generate statement to select a component according to the implementation, you will certainly get warnings.

--warn-library
Warns if a design unit replaces another design unit with the same name.

--warn-vital-generic
Warns if a generic name of a vital entity is not a vital generic name. This is set by default.

--warn-delayed-checks
Warns for checks that cannot be done during analysis time and are postponed to elaboration time. This is because not all procedure bodies are available during analysis (either because a package body has not yet been analysed or because *GHDL* doesn't read not required package bodies).

These are checks for no wait statement in a procedure called in a sensitized process and checks for pure rules of a function.

--warn-body
Emit a warning if a package body which is not required is analyzed. If a package does not declare a subprogram or a deferred constant, the package does not require a body.

--warn-specs
Emit a warning if an all or others specification does not apply.

--warn-unused
Emit a warning when a subprogram is never used.

--warn-error
When this option is set, warnings are considered as errors.

--warn-nested-comment
Emit a warning if a /* appears within a block comment (vhdl 2008).

--warn-parenthesis
Emit a warning in case of weird use of parenthesis

--warn-runtime-error
Emit a warning in case of runtime error that is detected during analysis.

Rebuilding commands

Analyzing and elaborating a design consisting in several files can be tricky, due to dependencies. GHDL has a few commands to rebuild a design.

Import command

Add files in the work design library:

```
ghdl -i [options] file...
```

All the files specified in the command line are scanned, parsed and added in the libraries but as not yet analyzed. No object files are created.

The purpose of this command is to localize design units in the design files. The make command will then be able to recursively build a hierarchy from an entity name or a configuration name.

Since the files are parsed, there must be correct files. However, since they are not analyzed, many errors are tolerated by this command.

Note that all the files are added to the work library. If you have many libraries, you must use the command for each library.

See *Make command*, to actually build the design.

Make command

Analyze automatically outdated files and elaborate a design:

```
ghdl -m [options] primary [secondary]
```

The primary unit denoted by the `primary` argument must already be known by the system, either because you have already analyzed it (even if you have modified it) or because you have imported it. GHDL analyzes all outdated files. A file may be outdated because it has been modified (e.g. you just have edited it), or because a design unit contained in the file depends on a unit which is outdated. This rule is of course recursive.

With the @code{-b} (bind only) option, GHDL will stop before the final linking step. This is useful when the main entry point is not GHDL and you're linking GHDL object files into a foreign program.

With the `-f` (force) option, GHDL analyzes all the units of the work library needed to create the design hierarchy. Not outdated units are recompiled. This is useful if you want to compile a design hierarchy with new compilation flags (for example, to add the *-g* debugging option).

The make command will only re-analyze design units in the work library. GHDL fails if it has to analyze an outdated unit from another library.

The purpose of this command is to be able to compile a design without prior knowledge of file order. In the VHDL model, some units must be analyzed before others (e.g. an entity before its architecture). It might be a nightmare to analyze a full design of several files, if you don't have the ordered list of file. This command computes an analysis order.

The make command fails when a unit was not previously parsed. For example, if you split a file containing several design units into several files, you must either import these new files or analyze them so that GHDL knows in which file these units are.

The make command imports files which have been modified. Then, a design hierarchy is internally built as if no units are outdated. Then, all outdated design units, using the dependencies of the design hierarchy, are analyzed. If necessary, the design hierarchy is elaborated.

This is not perfect, since the default architecture (the most recently analyzed one) may change while outdated design files are analyzed. In such a case, re-run the make command of GHDL.

Generate Makefile command

Generate a Makefile to build a design unit:

```
ghdl --gen-makefile [options] primary [secondary]
```

This command works like the make command (see *Make command*), but only a makefile is generated on the standard output.

Library commands

GHDL has a few commands which act on a library.

Directory command

Display the name of the units contained in a design library:

```
ghdl --dir [options] [libs]
```

The directory command, selected with the *–dir* command line argument displays the content of the design libraries (by default the `work` library). All options are allowed, but only a few are meaningful: `--work=NAME`, `--workdir=PATH` and `--std=VER`.

Clean command

Remove object and executable files but keep the library:

```
ghdl --clean [options]
```

GHDL tries to remove any object, executable or temporary file it could have created. Source files are not removed.

There is no short command line form for this option to prevent accidental clean up.

Remove command

Do like the clean command but remove the library too:

```
ghdl --remove [options]
```

There is no short command line form for this option to prevent accidental clean up. Note that after removing a design library, the files are not known anymore by GHDL.

Copy command

Make a local copy of an existing library:

```
ghdl --copy --work=name [options]
```

Make a local copy of an existing library. This is very useful if you want to add unit to the `ieee` library:

```
ghdl --copy --work=ieee --ieee=synopsys
ghdl -a --work=ieee numeric_unsigned.vhd
```

Create a Library

A new library is created by compiling entities (packages etc.) into it:

```
ghdl -a --work=my_custom_lib my_file.vhd
```

A library's source code is usually stored and compiled into its own directory, that you specify with the *`--workdir`* option:

```
ghdl -a --work=my_custom_lib --workdir=my_custom_libdir my_custom_lib_srcdir/my_file.
↪vhd
```

See also the `-PPATH` command line option.

Cross-reference command

To easily navigate through your sources, you may generate cross-references:

```
ghdl --xref-html [options] file...
```

This command generates an html file for each `file` given in the command line, with syntax highlighting and full cross-reference: every identifier is a link to its declaration. Besides, an index of the files is created too.

The set of `file` are analyzed, and then, if the analysis is successful, html files are generated in the directory specified by the `-o dir` option, or `html/` directory by default.

If the option `--format=html2` is specified, then the generated html files follow the HTML 2.0 standard, and colours are specified with *<FONT>* tags. However, colours are hard-coded.

If the option `--format=css` is specified, then the generated html files follow the HTML 4.0 standard, and use the CSS-1 file `ghdl.css` to specify colours. This file is generated only if it does not already exist (it is never overwritten) and can be customized by the user to change colours or appearance. Refer to a generated file and its comments for more information.

File commands

The following commands act on one or several files. They do not analyze files, therefore, they work even if a file has semantic errors.

Pretty print command

Generate HTML on standard output from VHDL:

```
ghdl --pp-html [options] file...
```

The files are just scanned and an html file, with syntax highlighting is generated on standard output.

Since the files are not even parsed, erroneous files or incomplete designs can be pretty printed.

The style of the html file can be modified with the `--format=` option. By default or when the `--format=html2` option is specified, the output is an HTML 2.0 file, with colours set through *<FONT>* tags. When the `--format=css` option is specified, the output is an HTML 4.0 file, with colours set through a CSS file, whose name is `ghdl.css`. See *Cross-reference command*, for more details about this CSS file.

Find command

Display the name of the design units in files:

```
ghdl -f file...
```

The files are scanned, parsed and the names of design units are displayed. Design units marked with two stars are candidate to be at the apex of a design hierarchy.

Chop command

Chop (or split) files at design unit:

```
ghdl --chop files
```

GHDL reads files, and writes a file in the current directory for every design unit.

The filename of a design unit is build according to the unit. For an entity declaration, a package declaration or a configuration the file name is `NAME.vhdl`, where *NAME* is the name of the design unit. For a package body, the filename is `NAME-body.vhdl`. Finally, for an architecture *ARCH* of an entity *ENTITY*, the filename is `ENTITY-ARCH.vhdl`.

Since the input files are parsed, this command aborts in case of syntax error. The command aborts too if a file to be written already exists.

Comments between design units are stored into the most adequate files.

This command may be useful to split big files, if your computer has not enough memory to compile such files. The size of the executable is reduced too.

Lines command

Display on the standard output lines of files preceded by line number:

```
ghdl --lines files
```

Misc commands

There are a few GHDL commands which are seldom useful.

Help command

Display (on the standard output) a short description of the all the commands available. If the help switch is followed by a command switch, then options for this later command are displayed:

```
ghdl --help
ghdl -h
ghdl -h command
```

Disp config command

Display the program paths and options used by GHDL:

```
ghdl --disp-config [options]
```

This may be useful to track installation errors.

Disp standard command

Display the `std.standard` package:

```
ghdl --disp-standard [options]
```

Version command

Display the *GHDL* version and exit:

```
ghdl --version
```

VPI build commands

These commands simplify the compile and the link of a user vpi module. They are all wrapper: the arguments are in fact a whole command line that is executed with additional switches. Currently a unix-like compiler (like *cc*, *gcc* or *clang*) is expected: the additional switches use their syntax. The only option is *-v* which displays the command before its execution.

VPI compile command

Add include path to the command and execute it:

```
ghdl --vpi-compile command
```

This will execute:

```
command -Ixxx/include
```

For example:

```
ghdl --vpi-compile gcc -c vpi1.c
```

executes:

```
gcc -c vpi1.c -fPIC -Ixxx/include
```

VPI link command

Add library path and name to the command and execute it:

```
ghdl --vpi-link command
```

This will execute:

```
command -Lxxx/lib -lghdlvpi
```

For example:

```
ghdl --vpi-link gcc -o vpi1.vpi vpi1.o
```

executes:

```
gcc -o vpi1.vpi vpi1.o --shared -Lxxx/lib -lghdlvpi
```

VPI cflags command

Display flags added by `--vpi-compile`:

```
ghdl --vpi-cflags
```

VPI ldflags command

Display flags added by `--vpi-link`:

```
ghdl --vpi-ldflags
```

VPI include dir command

Display the include directory added by the compile flags:

```
ghdl --vpi-include-dir
```

VPI library dir command

Display the library directory added by the link flags:

```
ghdl --vpi-library-dir
```

Installation Directory

During analysis and elaboration *GHDL* may read the *std* and *ieee* files. The location of these files is based on the prefix, which is (in priority order):

- the `--PREFIX=` command line option
- the `GHDL_PREFIX` environment variable
- a built-in default path. It is a hard-coded path on GNU/Linux and the value of the `HKLMSoftwareGhdlInstall_Dir` registry entry on Windows.

You should use the `--disp-config` command (*Disp config command* for details) to disp and debug installation problems.

IEEE library pitfalls

When you use options `--ieee=synopsys` or `--ieee=mentor`, the *IEEE* library contains non standard packages such as `std_logic_arith`.

These packages are not standard because there are not described by an IEEE standard, even if they have been put in the *IEEE* library. Furthermore, they are not really de-facto standard, because there are slight differences between the packages of Mentor and those of Synopsys.

Furthermore, since they are not well-thought, their use has pitfalls. For example, this description has error during compilation:

```
library ieee;
use ieee.std_logic_1164.all;

--  A counter from 0 to 10.
entity counter is
   port (val : out std_logic_vector (3 downto 0);
         ck : std_logic;
         rst : std_logic);
end counter;

library ieee;
use ieee.std_logic_unsigned.all;

architecture bad of counter
is
   signal v : std_logic_vector (3 downto 0);
begin
   process (ck, rst)
   begin
     if rst = '1' then
        v <= x"0";
     elsif rising_edge (ck) then
        if v = "1010" then -- Error
           v <= x"0";
        else
           v <= v + 1;
        end if;
     end if;
   end process;
```

```
   val <= v;
end bad;
```

When you analyze this design, GHDL does not accept it (too long lines have been split for readability):

```
ghdl -a --ieee=synopsys bad_counter.vhdl
bad_counter.vhdl:13:14: operator "=" is overloaded
bad_counter.vhdl:13:14: possible interpretations are:
../../libraries/ieee/std_logic_1164.v93:69:5: implicit function "="
    [std_logic_vector, std_logic_vector return boolean]
../../libraries/synopsys/std_logic_unsigned.vhdl:64:5: function "="
    [std_logic_vector, std_logic_vector return boolean]
../translate/ghdldrv/ghdl: compilation error
```

Indeed, the *"="* operator is defined in both packages, and both are visible at the place it is used. The first declaration is an implicit one, which occurs when the *std_logic_vector* type is declared and is an element to element comparison, the second one is an explicit declared function, with the semantic of an unsigned comparison.

With some analyser, the explicit declaration has priority over the implicit declaration, and this design can be analyzed without error. However, this is not the rule given by the VHDL LRM, and since GHDL follows these rules, it emits an error.

You can force GHDL to use this rule with the *-fexplicit* option. *GHDL options*, for more details.

However it is easy to fix this error, by using a selected name:

```
library ieee;
use ieee.std_logic_unsigned.all;

architecture fixed_bad of counter
is
   signal v : std_logic_vector (3 downto 0);
begin
   process (ck, rst)
   begin
     if rst = '1' then
        v <= x"0";
     elsif rising_edge (ck) then
        if ieee.std_logic_unsigned."=" (v, "1010") then
           v <= x"0";
        else
           v <= v + 1;
        end if;
     end if;
   end process;

   val <= v;
end fixed_bad;
```

It is better to only use the standard packages defined by IEEE, which provides the same functionalities:

```
library ieee;
use ieee.numeric_std.all;

architecture good of counter
is
   signal v : unsigned (3 downto 0);
begin
   process (ck, rst)
```

```
   begin
     if rst = '1' then
        v <= x"0";
     elsif rising_edge (ck) then
        if v = "1010" then
           v <= x"0";
        else
           v <= v + 1;
        end if;
     end if;
   end process;

   val <= std_logic_vector (v);
end good;
```

IEEE math packages

The `ieee` math packages (`math_real` and `math_complex`) provided with *GHDL* are fully compliant with the *IEEE* standard.

CHAPTER 4

Simulation and runtime

Simulation options

In most system environments, it is possible to pass options while invoking a program. Contrary to most programming languages, there is no standard method in VHDL to obtain the arguments or to set the exit status.

In GHDL, it is impossible to pass parameters to your design. A later version could do it through the generics interfaces of the top entity.

However, the GHDL runtime behaviour can be modified with some options; for example, it is possible to stop simulation after a certain time.

The exit status of the simulation is `EXIT_SUCCESS` (0) if the simulation completes, or `EXIT_FAILURE` (1) in case of error (assertion failure, overflow or any constraint error).

Here is the list of the most useful options. Some debugging options are also available, but not described here. The *`--help`* options lists all options available, including the debugging one.

`--assert-level`=<LEVEL>
: Select the assertion level at which an assertion violation stops the simulation. *LEVEL* is the name from the *severity_level* enumerated type defined in the *standard* package or the `none` name.

  By default, only assertion violation of severity level `failure` stops the simulation.

  For example, if *LEVEL* was `warning`, any assertion violation with severity level `warning`, `error` or `failure` would stop simulation, but the assertion violation at the `note` severity level would only display a message.

  Option `--assert-level=none` prevents any assertion violation to stop simulation.

`--ieee-asserts`=<POLICY>
: Select how the assertions from `ieee` units are handled. *POLICY* can be `enable` (the default), `disable` which disables all assertion from `ieee` packages and `disable-at-0` which disables only at start of simulation.

  This option can be useful to avoid assertion message from `ieee.numeric_std` (and other `ieee` packages).

--stop-time=<TIME>
Stop the simulation after TIME. TIME is expressed as a time value, *without* any space. The time is the simulation time, not the real clock time.

For example:

```
$ ./my_design --stop-time=10ns
$ ./my_design --stop-time=ps
```

--stop-delta=<N>
Stop the simulation after *N* delta cycles in the same current time.

--disp-time
Display the time and delta cycle number as simulation advances.

--disp-tree[=<KIND>]
Display the design hierarchy as a tree of instantiated design entities. This may be useful to understand the structure of a complex design. *KIND* is optional, but if set must be one of:

- none Do not display hierarchy. Same as if the option was not present.
- inst Display entities, architectures, instances, blocks and generates statements.
- proc Like inst but also display processes.
- port Like proc but display ports and signals too. If *KIND* is not specified, the hierarchy is displayed with the port mode.

--no-run
Do not simulate, only elaborate. This may be used with *--disp-tree* to display the tree without simulating the whole design.

--unbuffered
Disable buffering on stdout, stderr and files opened in write or append mode (TEXTIO).

--read-wave-opt=<FILENAME>
Filter signals to be dumped to the wave file according to the wave option file provided.

Here is a description of the wave option file format currently supported :

$ version = 1.1 # Optional

Path format for signals in packages : my_pkg.global_signal_a

Path format for signals in entities : /top/sub/clk

Dumps every signals named reset in first level sub entities of top /top/*/reset

Dumps every signals named reset in recursive sub entities of top /top/**/reset

Dump every signals of sub2 which could be anywhere in design except on # top level /**/sub2/*

Dump every signals of sub3 which must be a first level sub entity of the # top level *//sub3/*

Dump every signals of the first level sub entities of sub3 (but not # those of sub3) /**/sub3//

--write-wave-opt=<FILENAME>
If the wave option file doesn't exist, creates it with all the signals of the design. Otherwise throws an error, because it won't erase an existing file.

--vcd=<FILENAME>

--vcdgz=<FILENAME>
Option *--vcd* dumps into the VCD file *FILENAME* the signal values before each non-delta cycle. If *FILENAME* is -, then the standard output is used, otherwise a file is created or overwritten.

The `--vcdgz` option is the same as the *–vcd* option, but the output is compressed using the *zlib* (*gzip* compression). However, you can't use the - filename. Furthermore, only one VCD file can be written.

VCD (value change dump) is a file format defined by the *verilog* standard and used by virtually any wave viewer.

Since it comes from *verilog*, only a few VHDL types can be dumped. GHDL dumps only signals whose base type is of the following:

- types defined in the `std.standard` package:
- `bit`
- `bit_vector`
- types defined in the `ieee.std_logic_1164` package:
- `std_ulogic`
- `std_logic` (because it is a subtype of `std_ulogic`)
- `std_ulogic_vector`
- `std_logic_vector`
- any integer type

I have successfully used *gtkwave* to view VCD files.

Currently, there is no way to select signals to be dumped: all signals are dumped, which can generate big files.

It is very unfortunate there is no standard or well-known wave file format supporting VHDL types. If you are aware of such a free format, please mail me (*Reporting bugs*).

--fst=<FILENAME>
: Write the waveforms into a *fst*, that can be displayed by *gtkwave*. The *fst* files are much smaller than VCD or *GHW* files, but it handles only the same signals as the VCD format.

--wave=<FILENAME>
: Write the waveforms into a *ghw* (GHdl Waveform) file. Currently, all the signals are dumped into the waveform file, you cannot select a hierarchy of signals to be dumped.

  The format of this file was defined by myself and is not yet completely fixed. It may change slightly. The `gtkwave` tool can read the GHW files.

  Contrary to VCD files, any VHDL type can be dumped into a GHW file.

--psl-report=<FILENAME>
: Write a report for PSL assertions and coverage at the end of simulation. The file is written using the JSON format, but still being human readable.

--sdf=<PATH>=<FILENAME>
: Do VITAL annotation on *PATH* with SDF file `FILENAME`.

  PATH is a path of instances, separated with `.` or `/`. Any separator can be used. Instances are component instantiation labels, generate labels or block labels. Currently, you cannot use an indexed name.

  Specifying a delay:

```
--sdf=min=<PATH>=<FILENAME>
--sdf=typ=<PATH>=<FILENAME>
--sdf=max=<PATH>=<FILENAME>
```

  If the option contains a type of delay, that is `min=`, `typ=` or `max=`, the annotator use respectively minimum, typical or maximum values. If the option does not contain a type of delay, the annotator use the typical delay.

  See *Backannotation*, for more details.

--help
Display a short description of the options accepted by the runtime library.

Debugging VHDL programs

Debugging VHDL programs using *GDB* is possible only on GNU/Linux systems.

GDB is a general purpose debugger for programs compiled by *GCC*. Currently, there is no VHDL support for *GDB*. It may be difficult to inspect variables or signals in *GDB*, however, *GDB* is still able to display the stack frame in case of error or to set a breakpoint at a specified line.

GDB can be useful to precisely catch a runtime error, such as indexing an array beyond its bounds. All error check subprograms call the *__ghdl_fatal* procedure. Therefore, to catch runtime error, set a breakpoint like this:

```
(gdb) break __ghdl_fatal
```

When the breakpoint is hit, use the *where* or *bt* command to display the stack frames.

CHAPTER 5

GHDL implementation of VHDL

This chapter describes several implementation defined aspect of VHDL in GHDL.

VHDL standards

This is very unfortunate, but there are many versions of the VHDL language, and they aren't backward compatible.

The VHDL language was first standardized in 1987 by IEEE as IEEE 1076-1987, and is commonly referred as VHDL-87. This is certainly the most important version, since most of the VHDL tools are still based on this standard.

Various problems of this first standard have been analyzed by experts groups to give reasonable ways of interpreting the unclear portions of the standard.

VHDL was revised in 1993 by IEEE as IEEE 1076-1993. This revision is still well-known.

Unfortunately, VHDL-93 is not fully compatible with VHDL-87, i.e. some perfectly valid VHDL-87 programs are invalid VHDL-93 programs. Here are some of the reasons:

- the syntax of file declaration has changed (this is the most visible source of incompatibility),
- new keywords were introduced (group, impure, inertial, literal, postponed, pure, reject, rol, ror, shared, sla, sll, sra, srl, unaffected, xnor),
- some dynamic behaviours have changed (the concatenation is one of them),
- rules have been added.

Shared variables were replaced by protected types in the 2000 revision of the VHDL standard. This modification is also known as 1076a. Note that this standard is not fully backward compatible with VHDL-93, since the type of a shared variable must now be a protected type (there was no such restriction before).

Minors corrections were added by the 2002 revision of the VHDL standard. This revision is not fully backward compatible with VHDL-00 since, for example, the value of the *'instance_name* attribute has slightly changed.

The latest version is 2008. Many features have been added, and GHDL doesn't implement all of them.

You can select the VHDL standard expected by GHDL with the `--std=VER` option, where `VER` is one of the left column of the table below:

87 Select VHDL-87 standard as defined by IEEE 1076-1987. LRM bugs corrected by later revisions are taken into account.

93 Select VHDL-93; VHDL-87 file declarations are not accepted.

93c Select VHDL-93 standard with relaxed rules:

- VHDL-87 file declarations are accepted;
- default binding indication rules of VHDL-02 are used. Default binding rules are often used, but they are particularly obscure before VHDL-02.

00 Select VHDL-2000 standard, which adds protected types.

02 Select VHDL-2002 standard

08 Select VHDL-2008 standard (partially implemented).

The 93, 93c, 00 and 02 standards are considered as compatible: you can elaborate a design mixing these standards. However, 87, 93 and 08 are not compatible.

PSL implementation

GHDL understands embedded PSL annotations in VHDL files, but not in separate files.

As PSL annotations are embedded within comments, you must analyze and elaborate your design with option *-fpsl* to enable PSL annotations.

A PSL assertion statement must appear within a comment that starts with the *psl* keyword. The keyword must be followed (on the same line) by a PSL keyword such as *assert* or *default*. To continue a PSL statement on the next line, just start a new comment.

A PSL statement is considered as a process. So it is not allowed within a process.

All PSL assertions must be clocked (GHDL doesn't support unclocked assertion). Furthermore only one clock per assertion is allowed.

You can either use a default clock like this:

```
-- psl default clock is rising_edge (CLK);
-- psl assert always
--   a -> eventually! b;
```

or use a clocked expression (note the use of parenthesis):

```
-- psl assert (always a -> next[3](b)) @rising_edge (clk);
```

Of course only the simple subset of PSL is allowed.

Currently the built-in functions are not implemented.

Source representation

According to the VHDL standard, design units (i.e. entities, architectures, packages, package bodies and configurations) may be independently analyzed.

Several design units may be grouped into a design file.

In GHDL, a system file represents a design file. That is, a file compiled by GHDL may contain one or more design units.

It is common to have several design units in a design file.

GHDL does not impose any restriction on the name of a design file (except that the filename may not contain any control character or spaces).

GHDL do not keep a binary representation of the design units analyzed like other VHDL analyzers. The sources of the design units are re-read when needed (for example, an entity is re-read when one of its architecture is analyzed). Therefore, if you delete or modify a source file of a unit analyzed, GHDL will refuse to use it.

Library database

Each design unit analyzed is placed into a design library. By default, the name of this design library is `work`; however, this can be changed with the `--work=NAME` option of GHDL.

To keep the list of design units in a design library, GHDL creates library files. The name of these files is `NAME-objVER.cf`, where *NAME* is the name of the library, and *VER* the VHDL version (87, 93 or 08) used to analyze the design units.

You don't have to know how to read a library file. You can display it using the *-d* of *ghdl*. The file contains the name of the design units, as well as the location and the dependencies.

The format may change with the next version of GHDL.

Top entity

There are some restrictions on the entity being at the apex of a design hierarchy:

- The generic must have a default value, and the value of a generic is its default value;
- The ports type must be constrained.

Using vendor libraries

Many vendors libraries have been analyzed with GHDL. There are usually no problems. Be sure to use the `--work=` option. However, some problems have been encountered.

GHDL follows the VHDL LRM (the manual which defines VHDL) more strictly than other VHDL tools. You could try to relax the restrictions by using the `--std=93c`, *`-fexplicit`*, *`-frelaxed-rules`* and `--warn-no-vital-generic`.

Interfacing to other languages

Interfacing with foreign languages is possible only on GNU/Linux systems.

You can define a subprogram in a foreign language (such as *C* or *Ada*) and import it in a VHDL design.

Foreign 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 made it empty or simply but 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.

Restrictions on foreign 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 of non-composite types are easily imported:

integer types They are represented on a 32 bits word. This generally corresponds to *int* for *C* or *Integer* for *Ada*.

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

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

enumeration types They are represented on 8 bits or 32 bits word, if the number of literals is greater than 256. There is no corresponding C types, since arguments are not promoted.

Non-composite types are passed by value. For the *in* mode, 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 *in* 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 array are represented by a fat pointer. Do not use unconstrained arrays in foreign subprograms.

Accesses to an unconstrained array is a fat pointer. Other accesses correspond to an address and are passed to a subprogram like other non-composite types.

Files are represented by a 32 bits word, which corresponds to an index in a table.

Linking with foreign object files

You may add additional files or options during the link using the *-Wl,* of *GHDL*, as described in *Elaboration command*. 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.

Starting a simulation from a foreign program

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");
```

This function must be called once, and returns 0 at the end of the simulation. In case of failure, this function does not return. This has to be fixed.

Linking with Ada

As explained previously in *Starting a simulation from a foreign program*, you can start a simulation from an *Ada* program. However the build process is not trivial: you have to elaborate your *Ada* program and your *VHDL* design.

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 *Ada* program at link it with your *VHDL* design:

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

Using GRT from Ada

Warning: This topic is only for advanced users knowing how to use *Ada* and *GNAT*. This is provided only for reference, I 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.

For example, 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.
   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

GHDL implementation of VITAL

This chapter describes how VITAL is implemented in GHDL. Support of VITAL is really in a preliminary stage. Do not expect too much of it as now.

VITAL packages

The VITAL standard or IEEE 1076.4 was first published in 1995, and revised in 2000.

The version of the VITAL packages depends on the VHDL standard. VITAL 1995 packages are used with the VHDL 1987 standard, while VITAL 2000 packages are used with other standards. This choice is based on the requirements of VITAL: VITAL 1995 requires the models follow the VHDL 1987 standard, while VITAL 2000 requires the models follow VHDL 1993.

The VITAL 2000 packages were slightly modified so that they conform to the VHDL 1993 standard (a few functions are made pure and a few one impure).

VHDL restrictions for VITAL

The VITAL standard (partially) implemented is the IEEE 1076.4 standard published in 1995.

This standard defines restriction of the VHDL language usage on VITAL model. A *VITAL model* is a design unit (entity or architecture) decorated by the *VITAL_Level0* or *VITAL_Level1* attribute. These attributes are defined in the *ieee.VITAL_Timing* package.

Currently, only VITAL level 0 checks are implemented. VITAL level 1 models can be analyzed, but GHDL doesn't check they comply with the VITAL standard.

Moreover, GHDL doesn't check (yet) that timing generics are not read inside a VITAL level 0 model prior the VITAL annotation.

The analysis of a non-conformant VITAL model fails. You can disable the checks of VITAL restrictions with the *–no-vital-checks*. Even when restrictions are not checked, SDF annotation can be performed.

Backannotation

Backannotation is the process of setting VITAL generics with timing information provided by an external files.

The external files must be SDF (Standard Delay Format) files. GHDL supports a tiny subset of SDF version 2.1, other version number can be used, provided no features added by the next version are used.

Hierarchical instance names are not supported. However you can use a list of instances. If there is no instance, the top entity will be annotated and the celltype must be the name of the top entity. If there is at least one instance, the last instance name must be a component instantiation label, and the celltype must be the name of the component declaration instantiated.

Instances being annotated are not required to be VITAL compliant. However generics being annotated must follow rules of VITAL (e.g., type must be a suitable vital delay type).

Currently, only timing constraints applying on a timing generic of type *VitalDelayType01* has been implemented. This SDF annotator is just a proof of concept. Features will be added with the following GHDL release.

Negative constraint calculation

Negative constraint delay adjustment are necessary to handle negative constraint such as a negative setup time. This step is defined in the VITAL standard and should occur after backannotation.

GHDL does not do negative constraint calculation. It fails to handle models with negative constraint. I hope to be able to add this phase soon.

CHAPTER 7

Flaws and bugs report

Despite all the testing and already reported issues, you can find bugs or propose enhancements.

Reporting bugs

In order to improve GHDL, we welcome bugs report and suggestions for any aspect of GHDL. Please create an issue on https://github.com/tgingold/ghdl/issues

If the compiler crashes, this is a bug. Reliable tools never crash.

If your compiled VHDL executable crashes, this may be a bug at runtime or the code produced may be wrong. However, since VHDL has a notion of pointers, an erroneous VHDL program (using invalid pointers for example) may crash.

If the compiler emits an error message for a perfectly valid input or does not emit an error message for an invalid input, this may be a bug. Please send the input file and what you expected. If you know the LRM well enough, please specify the paragraph which has not been well implemented. If you don't know the LRM, maybe your bug report will be rejected simply because there is no bug. In the latter case, it may be difficult to discuss the issue; and comparisons with other VHDL tools is not a very strong argument.

If a compiler message is not clear enough for you, please tell me. The error messages can be improved, but I have not enough experience with them.

If you have found a mistake in the manual, please send a comment. If you have not understood some parts of this manual, please tell me. English is not my mother tongue, so this manual may not be well-written. Again, rewriting part of it is a good way to improve it.

If you send a *VHDL* file producing a bug, it is a good idea to try to make it as short as possible. It is also a good idea to make it looking like a test: write a comment which explains whether the file should compile, and if yes, whether or not it should run successfully. In the latter case, an assert statement should finish the test; the severity level note indicates success, while a severity level failure indicates failure.

For bug reports, please include enough information for the maintainers to reproduce the problem. This includes:

- the version of *GHDL* (you can get it with `ghdl --version`).
- the operating system
- whether you have built *GHDL* from sources or used the binary distribution.
- the content of the input files
- a description of the problem and samples of any erroneous input
- anything else that you think would be helpful.

Future improvements

I have several axes for *GHDL* improvements:

- Documentation.
- Better diagnostics messages (warning and error).
- Full support of VHDL-2008.
- Optimization (simulation speed).
- Graphical tools (to see waves and to debug)
- Style checks
- VITAL acceleration

CHAPTER 8

Copyrights

The GHDL front-end, the `std.textio` package and the runtime library (`grt`) are copyrighted Tristan Gingold, come with **absolutely no warranty**, and are distributed under the conditions of the General Public License.

The `ieee.numeric_bit` and `ieee.numeric_std` packages are copyrighted by the IEEE. The source files may be distributed without change, except as permitted by the standard.

This source file may not be sold or distributed for profit. See the source file and the IEEE 1076.3 standard for more information.

The `ieee.std_logic_1164`, `ieee.Math_Real` and `ieee.Math_Complex` packages are copyrighted by the IEEE. See source files for more information.

The `ieee.VITAL_Primitives`, `ieee.VITAL_Timing` and `ieee.VITAL_Memory` packages are copyrighted by IEEE. See source file and the IEEE 1076.4 standards for more information.

The packages `std_logic_arith`, `std_logic_signed`, `std_logic_unsigned` and `std_logic_textio` contained in the `synopsys` directory are copyrighted by Synopsys, Inc. The source files may be used and distributed without restriction provided that the copyright statements are not removed from the files and that any derivative work contains the copyright notice. See the source files for more information.

The package `std_logic_arith` contained in the `mentor` directory is copyrighted by Mentor Graphics. The source files may be distributed in whole without restriction provided that the copyright statement is not removed from the file and that any derivative work contains this copyright notice. See the source files for more information.

As a consequence of the runtime copyright, you may not be allowed to distribute an executable produced by *GHDL* without the VHDL sources. To my mind, this is not a real restriction, since there is no points in distributing VHDL executable. Please, send a comment (*Reporting bugs*) if you don't like this policy.

CHAPTER 9

Indices and tables

- genindex
- search

Index

Symbols

Numbers

A

B

C