GHDL Documentation

Release 1.0-dev

Tristan Gingold and contributors

Apr 08, 2020

Introduction

03.03.2019 - GHDL v0.36 was released.

23.02.2019 - GHDL v0.36-rc1 was released.

29.11.2018 - GHDL 20181129 was released.

20.12.2017 - A new GitHub organization was created.

14.12.2017 - GHDL 0.35 was released.

15.08.2017 - GHDL 0.34 was released.

23.10.2015 - GHDL 0.33 was released.

CHAPTER 1

What is *VHDL*?

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

Although VHDL was not designed for writing general purpose programs, VHDL *is* a programming language, and you can write any algorithm with it. 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*. Indeed, 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*. 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. At the same time, like a design written in another *HDL*, a set of VHDL sources can be transformed with a *synthesis tool* into a netlist, that is, a detailed gate-level implementation.

The development of VHDL started in 1983 and the standard is named IEEE *1076*. Four revisions exist: 1987, 1993, 2002 and 2008. The standardization is handled by the VHDL Analysis and Standardization Group (VASG/P1076).

CHAPTER 2

What is GHDL?

GHDL is a shorthand for *G Hardware Design Language* (currently, *G* has no meaning). It is a VHDL analyzer, compiler and simulator that can execute (nearly) any VHDL program. GHDL is *not* a synthesis tool: you cannot create a netlist with GHDL (yet).

Unlike some other simulators, GHDL is a compiler: it directly translates a VHDL file to machine code, 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.

GHDL can use multiple back-ends, i.e. code generators, (GCC, LLVM or x86/i386 only, a built-in one) and runs on GNU/Linux, Windows ™ and macOS ™ , both on x86 and on x86_64.

The current version of GHDL does not contain any built-in graphical viewer: you cannot see signal waves. You can still check the behavior of your design with a test bench. Moreover, the current version can produce GHW, VCD or *FST* files which can be viewed with a waveform viewer, such as GtkWave.

GHDL aims at implementing VHDL as defined by IEEE 1076. It supports the 1987, 1993 and 2002 revisions and, partially, the latest, 2008. PSL is also partially supported.

Several third party projects are supported: VUnit, OSVVM, cocotb (through the VPI interface), ...

Hint: Although synthesis is not available yet, there is some experimental support. See *Synthesis* for further info.

CHAPTER 3

Who uses GHDL?

CHAPTER 4

Contributing

The first step might be to use GHDL and explore its possibilities in your own project. If you are new to VHDL, see the *Quick Start Guide* for an introduction. Furthermore, we encourage you to read *Invoking GHDL*, where the most commonly used options are explained. You can also check the complete *Command Reference*.

If you are more familiar with GHDL, you might start asking yourself how it works internally. If so, you might find *Implementation of VHDL* and *Implementation of VITAL* interesting.

While using GHDL, you might find flaws, such as bugs, missing features, typos in the documentation, or topics which still are not covered. In order to improve GHDL, we welcome bug reports, suggestions, and contributions for any aspect of GHDL. Whether it's a bug or an enhancement, have a look at the and to see if someone already told us about it. You might find a solution there.

If you found no information on your topic, please, report so that we are aware! You can reach us through various ways:

Hint: Since the development of GHDL started fifteen years ago, multiple platforms have been used as a support for both distribution and getting feedback. However, the development is now centralized in github.

Tip: How To Ask Questions The Smart Way

4.1 Reporting bugs

Tip:

- If the compiler crashes, this is a bug. Reliable tools never 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.
- If the executable created from your VHDL sources crashes, this may be a bug at runtime or the code itself may be wrong. However, since VHDL has a notion of pointers, an erroneous VHDL program (using invalid pointers for example) may crash.

- If a compiler message is not clear enough, please tell us. The error messages can be improved, but we do not have enough experience with them.

Please, report issues of this kind through , as this allows us to categorize issues into groups and to assign developers to them. You can track the issue's state and see how it's getting solved.

Important: As suggested in the bug report template, please elaborate a *Minimal (non) Working Example* (MWE) prior to sending the report, so that the possible bug source is isolated. Should it fulfill the format requirements of issue-runner, you would be able to test your bug with the latest GHDL version. Please do so in order to ensure that the bug is not solved already.

Also, please include enough information in the bug report, for the maintainers to reproduce the problem. The template includes:

- Operating system and version of GHDL (you can get it with `ghdl --version`).
- Whether you have built GHDL from sources (provide short SHA of the used commit) or used the binary distribution (note which release/tag).
  - If you cannot compile, please report which compiler you are using and the version.
- Content of the input files which comprise the MWE
- Description of the problem:
  - Comment explaining whether the MWE should compile or not; if yes, whether or not is should run until the assertion.
  - What you expect to happen and what you actually get. If you know the LRM well enough, please specify which paragraph might not be implemented well.
  - Samples of any log.
  - Anything else that you think would be helpful.

Note: If you don't know the LRM, be aware that an issue claimed as a bug report may be rejected because there is no bug according to it. GHDL aims at implementing VHDL as defined in IEEE 1076. However, some other tools allow constructs which do not fully follow the standard revisions. Therefore, comparisons with other VHDL is not a solid argument. Some of them are supported by GHDL (see *IEEE library pitfalls*), but any such enhancement will have very low priority.

4.2 Requesting enhancements

All enhancements and feature requests are welcome. Please open a new issue to report any, so you can track the request's status and implementation. Depending on the complexity of the request, you may want to chat on Gitter, to polish it before opening an issue.

4.3 Improving the documentation

If you found a mistake in the documentation, please send a comment. If you didn't understand some parts of this manual, please tell us. English is not our mother tongue, so this documentation may not be well-written.

Likewise, rewriting part of the documentation or missing content (such as examples) is a good way to improve it. Since it automatically is built from *reStructuredText* and *Markdown* sources, you can fork, modify and request the maintainers to pull your copy. See *Fork, modify and pull-request*.

4.4 Fork, modify and pull-request

Tip:

- Before starting any modification, you might want to have a look at and , to check which other contributions are being made or have been made. If you observe that the modifications you are about to start might conflict with any other, please
- See section *Directory structure* to faster find the location of the sources you need to modify, and/or to know where to place new ones.

Contributing source code/documentation via Git is very easy. Although we don't provide direct write access to our repositories, the project is hosted at GitHub, which follows a fork, edit and pull-request flow . That is:

1. Make a copy (fork) of the project.
2. Do the changes you wish (edit, add, rename, move and/or delete).
3. When you think that the changes are ready to be merged, notify the maintainers by opening a Pull Request (PR).
4. The maintainers will review the proposed changes and will reply in the corresponding thread if any further modification is required. If so, you can keep adding commits to the same branch, and the PR will be automatically updated.
5. Last, the maintainers will merge your branch. You will be notified, the PR will be closed, and you'll be allowed to delete the branch, if you want.

Tip:

- It is recommended to read A successful Git branching model for a reference on how maintainers expect to handle multiple branches. However, our actual model is not as exhaustive as explained there.
- Some commit messages can automatically close issues. This is a very useful feature, which you are not required to use. However beware that using *fix* anywhere in the commit message can have side effects. If you closed any issue unexpectedly, just reply to it (even if it's closed) so that maintainers can check it.
- It is recommended to read *Coding Style* before contributing modifications to Ada sources.

4.5 Related interesting projects

If you have an interesting project, please send us feedback or get listed on our *Who uses GHDL?* page.

CHAPTER 5

Copyrights | Licenses

- The GHDL front-end package `std.textio`, and the runtime library `grt` are given under *GNU GPLv2*.
- The documentation is given under *CC-BY-SA*.

Warning: As a consequence of the runtime copyright, you are not allowed to distribute an executable produced by GHDL without the VHDL sources. To my mind, this is not a real restriction, since it is pointless to distribute VHDL executable. Please, send a comment (*Requesting enhancements*) if you don't like this policy.

- The following packages are copyrighted by third parties (see corresponding sources for more information):
  - These from library `ieee` are copyrighted by Institute of Electrical and Electronics Engineers (IEEE) :
    - `numeric_bit` and `numeric_std`: the source files may be distributed without change, except as permitted by the standard; these may not be sold or distributed for profit. [see also IEEE 1076.3]
    - `std_logic_1164`, `Math_Real` and `Math_Complex`
    - `VITAL_Primitives`, `VITAL_Timing` and `VITAL_Memory` [see also IEEE 1076.4]
  - The following sources 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.
    - `synopsys` directory: `std_logic_arith`, `std_logic_signed`, `std_logic_unsigned` and `std_logic_textio` are copyrighted by Synopsys, Inc.
    - `mentor` directory: `std_logic_arith` is copyrighted by Mentor Graphics

5.1 GNU GPLv2

5.2 CC-BY-SA

This is a free documentation; you can redistribute it and/or modify it under the terms of the Creative Commons Attribution-ShareAlike 4.0 license. You are free to **share** (copy and redistribute the material in any medium or format) and/or **adapt** (remix, transform, and build upon the material for any purpose, even commercially). We cannot revoke these freedoms as long as you follow the these terms:

- **Attribution**: you must provide the name of the creator and attribution parties (more info), a copyright notice, a license notice, a disclaimer notice, a link to the material, a link to the license and indicate if changes were made (see marking guide and more info). You may do so in any reasonable manner, but not in any way that suggests we endorse you or your use.
- **ShareAlike**: if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- **No additional restrictions**: you may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

See CC-BY-SA-4.0 Legal Code for more details.

5.3 List of Contributors

Contributor[1]	Role
Baggett, Jonas	signal selection
Bertram, Felix	VPI interface
Davis, Brian	Windows Mcode builds
Drummond, Brian	GCC 4.8.2 update, OSVVM port, some bugfixes
Gingold, Tristan[2]	**Sole author of GHDL as a whole**
Jensen, Adam	FreeBSD builds
Koch, Markus	vendor pre-compile script for Lattice (GNU/Linux)
Koontz, David	Mac OSX builds, LRM compliance work, bugfix analyses
Lehmann, Patrick	Windows compile scripts, vendor library pre-compile scripts (win+lin), building in MinGW, AppVeyor integration.
Martinez-Corral, Unai	Docker builds, Travis-CI & Docker, adapt/fix RTD theme
van Rantwijk, Joris	Debian packaging

Only those who made substantial contributions are shown in the table above, but many others contributed with minor patches. You can find a list at

With apologies to anyone who ought to be either on this table or in the GitHub contributor list, but isn't. Thanks also to all those who have reported bugs and support issues, and often patches and testcases to either the late gna! website or sourceforge.net/p/ghdl-updates/tickets.

[1] In alphabetical order
[2] Maintainer

Part I

Getting GHDL

CHAPTER 6

Releases and sources

Contents of this Page

6.1 Downloading pre-built packages

OS	Backend	Size	Downloads
buster	mcode-gpl.src	3.85 MB	v0.37/ghdl-0.37-buster-mcode-gpl.src.tgz
buster	mcode-gpl	2.68 MB	v0.37/ghdl-0.37-buster-mcode-gpl.tgz
buster	mcode-synth	3.57 MB	v0.37/ghdl-0.37-buster-mcode-synth.tgz
buster	mcode	3.01 MB	v0.37/ghdl-0.37-buster-mcode.tgz
fedora31	LLVM	6.55 MB	v0.37/ghdl-0.37-fedora31-llvm.tgz
fedora31	mcode	2.93 MB	v0.37/ghdl-0.37-fedora31-mcode.tgz
Max OS X	mcode	2.22 MB	v0.37/ghdl-0.37-macosx-mcode.tgz
Windows x86 (MinGW32)	mcode	4.89 MB	v0.37/ghdl-0.37-mingw32-mcode.zip
Windows x86 (MinGW64)	LLVM	19.99 MB	v0.37/ghdl-0.37-mingw64-llvm.zip
ubuntu16	llvm-3.9	6.51 MB	v0.37/ghdl-0.37-ubuntu16-llvm-3.9.tgz
ubuntu16	mcode	2.89 MB	v0.37/ghdl-0.37-ubuntu16-mcode.tgz
Sum:			v0.37

Pre-built packages of older releases

Release/Tag	Date
v0.36	2019-03-03
v0.36-rc1	2019-02-23
20181129	2018-11-29
v0.35	2017-12-14
v0.34	2017-08-15
2017-03-01	2017-03-01
2016-09-14	2016-09-14
2016-06-07	2016-06-07
2016-05-03	2016-05-03
v0.33	2015-11-18

6.2 Downloading Source Files

Hint: All the following procedures will retrieve the latest development version of GHDL, i.e., the *master* branch at github.com/ghdl/ghdl. We do our best to keep it stable, but bugs can seldom be published. See *HINT* boxes below for instructions to get older releases.

Tarball/zip-file

GHDL can be downloaded as a zip-file or tarball from GitHub. See the following table, to choose your desired format/version:

Hint: To download a specific version of GHDL, use this alternative URL, where `<format>` is `tar.gz` or `zip`: `https://codeload.github.com/ghdl/ghdl/<format>/<tag>`.

git clone

GHDL can be downloaded (cloned) with `git clone` from GitHub. GitHub offers the transfer protocols HTTPS and SSH. You should use SSH if you have a GitHub account and have already uploaded an OpenSSH public key to GitHub, otherwise use HTTPS if you have no account or you want to use login credentials.

Protocol	GitHub Repository URL
HTTPS	https://github.com/ghdl/ghdl.git
SSH	ssh://git@github.com:ghdl/ghdl.git

Hint: Execute `git checkout -b stable <tag>` after `git clone`, to checkout a specific version of GHDL.

Command line instructions to clone GHDL with HTTPS protocol:

```
cd GitRoot
git clone "https://github.com/ghdl/ghdl.git" ghdl
cd ghdl
git remote rename origin github
```

Command line instructions to clone GHDL with SSH protocol:

```
cd GitRoot
git clone "ssh://git@github.com:ghdl/ghdl.git" ghdl
cd ghdl
git remote rename origin github
```

Note: Executing the following instructions in Windows Command Prompt (**`cmd.exe`**) won't function or will result in errors! All Windows command line instructions are intended for **`Windows PowerShell`**, if not marked otherwise. **`Windows PowerShell`** can be installed or upgraded to v5.1 by installing the Windows Management Framework.

CHAPTER 7

Building GHDL from Sources

Download

GHDL can be downloaded as a zip-file/tar-file (latest 'master' branch) or cloned with `git clone` from GitHub. GitHub offers HTTPS and SSH as transfer protocols. See the *Downloading Source Files* page for further details.

Important: Since GHDL is written in *Ada*, independently of the code generator you use, the a compiler is required. Most GNU/Linux package managers provide a package named `gcc-ada` or `gcc-gnat`. Alternatively, *GNU Ada compiler*, *GNAT GPL*, can be downloaded anonymously from libre.adacore.com (2014, or later; for x86, 32 or 64 bits). Then, untar and run the doinstall script.

Available back-ends

GHDL currently supports three different back-ends (code generators):

- mcode - built-in x86 (or x86_64) code generator
- GCC - Gnu Compiler Collection (gcc.gnu.org)
- LLVM - Low-Level Virtual Machine (llvm.org)

Here is a short comparison, so that you can choose the one you want to use:

Back-end	Pros	Cons
mcode	• Very easy to build • Very quick analysis • Can handle very large designs	• Simulation is slower • x86_64/i386 only
LLVM	• Generated code is faster • Generated code can be debugged (with `-g`) • Easier to build than GCC • Ported to many platforms (x86, x86_64, armv7/aarch64)	• Build is more complex than mcode
GCC	• Generated code is faster (particularly with `-O` or `-O2`) • Generated code can be debugged (with `-g`) • Ported to many platforms (x86, x86_64, PowerPC, SPARC)	• Build is even more complex • Analysis can take time (particularly for large units) • Code coverage collection (`gcov`) is unique to GCC

7.1 Directory structure

- `src`: sources of GHDL, all of them in Ada.
- `libraries`: mostly third party libraries such as, *ieee*, *std*, *synopsys* and *vital*. Except for a few shell and *Python* scripts, all the content is written in VHDL.
  - Vendors like Altera, Lattice and Xilinx have their own simulation libraries, especially for FPGA primitives, soft and hard macros. These libraries cannot be shipped with GHDL, but we offer prepared compile scripts to pre-compile the vendor libraries, if the vendor tool is present on the computer. These are located in `libraries/vendor`. See *Precompile Vendor Primitives* for information on how to use them.
- `dist`: scripts and auxiliary files to build GHDL in different environments:
  - `gcc`: header and configuration files to build GHDL with GCC (all platforms).
  - `linux`: build and test script written in shell, and other auxiliary files used to i) launch docker containers and ii) automate multiple builds in Travis CI.
  - `windows`:
    * `mcode`:
    * `appveyor`:
- `doc`: *Markdown* and *reStructuredText* sources and auxiliary files to build the documentation with Sphinx. In fact, Read the Docs (RTD) is used to automatically build and deploy this site and/or PDF you are reading.
- `testsuite`: files used for testing.
- *.yml* configuration files for CI environments (`readthedocs`, `travis`, and `appveyor`) and *ignore* files for source control management tools (`git` and `.hg`).
- Files for building GHDL: `configure` and `Makefile.in`.

- Auxiliary files for development: `.gdbinit` and `ghdl.gpr.in` (GNAT project file).
- Text files: `COPYING.md`, `NEWS.md`, and `README.md`.

7.2 mcode backend

The mcode backend is available for all supported platforms and is also the simplest procedure, because it requires the fewest dependencies and configuration options.

7.2.1 GCC/GNAT: GNU/Linux or Windows (MinGW/MSYS2)

Requirements

- GCC (Gnu Compiler Collection)
- GNAT (Ada compiler for GCC)

GHDL is configured by `configure` and built by `make`.

- First, GHDL needs to be configured. It is common to specify a `PREFIX` (installation directory like `/usr/local` or `/opt/ghdl`). Without any other option, `configure` selects *mcode* as the backend.
- Next, `make` starts the compilation process.
- Finally, `make install` installs GHDL into the installation directory specified by `PREFIX`.

Hint: ON GNU/Linux, you may need super user privileges (`sudo ...`).

Example:

```
$ cd <ghdl>
$ mkdir build
$ cd build
$ ../configure --prefix=PREFIX
$ make
$ make install
```

7.2.2 GNAT GPL: Windows

Requirements

- GNAT GPL from http://libre.adacore.com
- PowerShell 4
- PowerShell Community Extensions (PSCX)

compile.ps1

7.3 LLVM backend

Requirements

- GCC (Gnu Compiler Collection)

- GNAT (Ada compiler for GCC)
- LLVM (Low-Level-Virtual Machine) and CLANG (Compiler front-end for LLVM): 3.5, 3.8, 3.9, 4.0, 5.0, 6.0, 7.0, 8.0 or 9.0

7.3.1 GCC/GNAT: GNU/Linux or Windows (MinGW/MSYS2)

Hint: You need to install LLVM (usually depends on `libedit`, see #29). Debugging is only supported with LLVM 3.5.

GHDL is configured by `configure` and built by `make`.

- First, GHDL needs to be configured. It is common to specify a `PREFIX` (installation directory like `/usr/local` or `/opt/ghdl`). Set the proper arg, `./configure --with-llvm-config`, to select LLVM backend. If `llvm-config` is not in your path, you can specify it: `./configure --with-llvm-config=LLVM_INSTALL/bin/llvm-config`.
- Next, `make` starts the compilation process.
- Finally, `make install` installs GHDL into the installation directory specified by `PREFIX`.

Example:

```
$ cd <ghdl>
$ mkdir build
$ cd build
$ ../configure --with-llvm-config --prefix=PREFIX
$ make
$ make install
```

Hint: If you want to have stack backtraces on errors (like assert failure or index of out bounds), you need to configure and build `libbacktrace` from GCC (you don't need to configure GCC). Then add the following arg to configure: `--with-backtrace-lib=/path-to-gcc-build/libbacktrace/.libs/libbacktrace.a`

7.4 GCC backend

Todo: Instructions to build GHDL with GCC backend on Windows are not available yet.

Requirements

- GCC (Gnu Compiler Collection)
- GNAT (Ada compiler for GCC)
- GCC source files. Download and untar the sources of version 4.9.x, 5.x, 6.x, 7.x, 8.x, 9.x or 10.x.

Hint: There are some dependencies for building GCC (`gmp`, `mpfr` and `mpc`). If you have not installed them on your system, you can either build them manually or use the `download_prerequisites` script provided in the GCC source tree (recommended): `cd /path/to/gcc/source/dir && ./contrib/download_prerequisites`.

- First configure GHDL, specify GCC source directory and installation prefix (like `/usr/local` or `/opt/ghdl`).
- Next, invoke `make copy-sources` to copy GHDL sources in the source directory.
- Then, configure GCC. The list of `--disable` configure options can be adjusted to your needs. GHDL does not require all these optional libraries and disabling them will speed up the build.
- Now, build and install GCC with `make`.
- Last, build and install GHDL libraries.

Example:

```
$ cd <ghdl>
$ mkdir build
$ cd build
$ ../configure --with-gcc=/path/to/gcc/source/dir --prefix=/usr/local
$ make copy-sources
$ mkdir gcc-objs; cd gcc-objs
$ /path/to/gcc/source/dir/configure --prefix=/usr/local --enable-languages=c,vhdl \
--disable-bootstrap --disable-lto --disable-multilib --disable-libssp \
--disable-libgomp --disable-libquadmath
$ make -j2 && make install
$ cd /path/to/ghdl/source/dir/build
$ make ghdllib
$ make install
```

Hint: Note that the prefix directory to configure `gcc` must be the same as the one used to configure GHDL. If you have manually built `gmp`/`mpfr`/`mpc` (without using the script in `contrib`), and, if you have installed them in a non-standard directory, you may need to add `--with-gmp=GMP_INSTALL_DIR`.

Hint: If your system gcc was configured with `--enable-default-pie` (check if that option appears in the output of `gcc -v`), you should also add it.

Hint: If you don't want to install `makeinfo`, do `make install MAKEINFO=true` instead.

Hint: Once GCC (with GHDL) has been built once, it is possible to work on the GHDL source tree without copying it in the GCC tree. Commands are:

```
$ make ghdl1-gcc           # Build the compiler
$ make ghdl_gcc            # Build the driver
$ make libs.vhdl.local_gcc # Compile the vhdl libraries
$ make grt-all             # Build the GHDL runtime
$ make install.vpi.local   # Locally install vpi files
```

In `src/ortho/gcc`, create a `Makefile.conf` file that sets the following variables:

```
AGCC_GCCSRC_DIR=/path/to/gcc/sources
AGCC_GCCOBJ_DIR=/path/to/gcc/build
```

If your system gcc was built with `--enable-default-pie`, add `-no-pie` option for linking.

Hint: For ppc64/ppc64le platform, the object file format contains an identifier for the source language. Because gcc doesn't know about VHDL, gcc crashes very early. This could be fixed with a very simple change in `gcc/config/rs6000/rs6000.c` (`gcc/config/rs6000/rs6000-logue.c` since gcc 10), function `rs6000_output_function_epilogue`:

```
          || ! strcmp (language_string, "GNU GIMPLE")
          || ! strcmp (language_string, "GNU Go")
          || ! strcmp (language_string, "GNU D")
-         || ! strcmp (language_string, "libgccjit"))
+         || ! strcmp (language_string, "libgccjit")
+         || ! strcmp (language_string, "vhdl"))
        i = 0;
```

Hint: The output of both GCC and LLVM is an executable file, but *mcode* does not generate any. Therefore, if using GCC/LLVM, the call with argument `-r` can be replaced with direct execution of the binary. See section *Quick Start Guide*.

After making your choice, you can jump to the corresponding section. However, we suggest you to read *Directory structure* first, so that you know where the content will be placed and which files are expected to be created.

Hint: In these instructions, the configure script is executed in the source directory; but you can execute in a different directory too, like this:

```
$ mkdir ghdl-objs
$ cd ghdl-objs
$ ../path/to/ghdl/configure ...
```

Hint: On Windows, building GHDL with mcode backend and GNAT GPL 32 bit seems to be the only way to get a standalone native executable.

- MINGW/MSYS2 builds depend on the environment/runtime.
- For 64 bit, no native compiler exists from AdaCore.
- That Ada to .NET compiler, which might work for 32 or 64 bit. is not up-to-date.

CHAPTER 8

Precompile Vendor Primitives

Vendors like Altera, Lattice and Xilinx have their own simulation libraries, especially for FPGA primitives, soft and hard macros. These libraries cannot be shipped with *GHDL*, but we offer prepared compile scripts to pre-compile the vendor libraries, if the vendor tool is present on the computer. There are also popular simulation and verification libraries like OSVVM[1] or UVVM[2], which can be pre-compiled, too.

The compilation scripts are writen in the shell languages: *PowerShell* for *Windows* ™ and *Bash* for *GNU/Linux*. The compile scripts can colorize the GHDL warning and error lines with the help of *grc/grcat*[3].

8.1 Supported Vendors Libraries

- Altera/Intel Quartus (13.0 or later):
  - *lpm*, *sgate*
  - *altera*, *altera_mf*, *altera_lnsim*
  - *arriaii*, *arriaii_pcie_hip*, *arriaiigz*
  - *arriav*, *arriavgz*, *arriavgz_pcie_hip*
  - *cycloneiv*, *cycloneiv_pcie_hip*, *cycloneive*
  - *cyclonev*
  - *max*, *maxii*, *maxv*
  - *stratixiv*, *stratixiv_pcie_hip*
  - *stratixv*, *stratixv_pcie_hip*
  - *fiftyfivenm*, *twentynm*
- Lattice (3.6 or later):
  - *ec*
  - *ecp*, *ecp2*, *ecp3*, *ecp5u*
  - *lptm*, *lptm2*

[1] OSVVM http://github.com/OSVVM/OSVVM
[2] UVVM https://github.com/UVVM/UVVM_All
[3] Generic Colourizer http://kassiopeia.juls.savba.sk/~garabik/software/grc.html

  - *machxo*, *machxo2*, *machxo3l*
  - *sc*, *scm*
  - *xp*, *xp2*
- Xilinx ISE (14.0 or later):
  - *unisim* (incl. *secureip*)
  - *unimacro*
  - *simprim* (incl. *secureip*)
  - *xilinxcorelib*
- Xilinx Vivado (2014.1 or later):
  - *unisim* (incl. *secureip*)
  - *unimacro*

8.2 Supported Simulation and Verification Libraries

- OSVVM (for VHDL-2008)
  - osvvm
- UVVM (for VHDL-2008)
  - uvvm-utilities
  - uvvm-vvc-framework
  - uvvm-vip-avalon_mm
  - uvvm-vip-axi_lite
  - uvvm-vip-axi_stream
  - uvvm-vip-gpio
  - uvvm-vip-i2c
  - uvvm-vip-sbi
  - uvvm-vip-spi
  - uvvm-vip-uart

8.3 Script Configuration

The vendor library compile scripts need to know where the used / latest vendor tool chain is installed. Therefore, the scripts implement a default installation directory search as well as environment variable checks. If a vendor tool cannot be detected or the script chooses the wrong vendor library source directory, then it's possible to provide the path via *–source* or *-Source*.

The generated output is stored relative to the current working directory. The scripts create a sub-directory for each vendor. The default output directory can be overwritten by the parameter *–output* or *-Output*.

To compile all source files with GHDL, the simulator executable is searched in *PATH*. The found default GHDL executable can be overwritten by setting the environment variable *GHDL* or by passing the parameter *–ghdl* or *-GHDL* to the scripts.

If the vendor library compilation is used very often, we recommend configuring these parameters in *config.sh* or *config.psm1*, so the command line can be shortened to the essential parts.

8.4 Compiling on Linux

- **Step 0 - Configure the scripts (optional)**

  See the next section for how to configure *config.sh*.

- **Step 1 - Browse to your simulation working directory**

```
$ cd <MySimulationFolder>
```

- **Step 2 - Start the compilation script(s)**

```
$ /usr/local/lib/ghdl/vendors/compile-altera.sh --all
$ /usr/local/lib/ghdl/vendors/compile-lattice.sh --all
$ /usr/local/lib/ghdl/vendors/compile-xilinx-ise.sh --all
$ /usr/local/lib/ghdl/vendors/compile-xilinx-vivado.sh --all
$ /usr/local/lib/ghdl/vendors/compile-osvvm.sh --all
$ /usr/local/lib/ghdl/vendors/compile-uvvm.sh --all
```

In most cases GHDL is installed into `/usr/local/`. The scripts are
installed into the `lib` directory.

- **Step 3 - Viewing the result**

  This creates vendor directories in your current working directory and compiles the vendor files into them.

```
$ ls -ahl
...
drwxr-xr-x  2 <user> <group>  56K Mar 09 17:41 altera
drwxr-xr-x  2 <user> <group>  56K Mar 09 17:42 lattice
drwxr-xr-x  2 <user> <group>  56K Mar 09 17:48 osvvm
drwxr-xr-x  2 <user> <group>  56K Mar 09 17:58 uvvm
drwxr-xr-x  2 <user> <group>  56K Mar 09 17:58 xilinx-ise
drwxr-xr-x  2 <user> <group>  56K Mar 09 17:48 xilinx-vivado
```

8.5 Compiling on Windows

- **Step 0 - Configure the scripts (optional)**

  See the next section for how to configure *config.psm1*.

- **Step 1 - Browse to your simulation working directory**

```
PS> cd <MySimulationFolder>
```

- **Step 2 - Start the compilation script(s)**

```
PS> <GHDL>\libraries\vendors\compile-altera.ps1 -All
PS> <GHDL>\libraries\vendors\compile-lattice.ps1 -All
PS> <GHDL>\libraries\vendors\compile-xilinx-ise.ps1 -All
PS> <GHDL>\libraries\vendors\compile-xilinx-vivado.ps1 -All
PS> <GHDL>\libraries\vendors\compile-osvvm.ps1 -All
PS> <GHDL>\libraries\vendors\compile-uvvm.ps1 -All
```

- **Step 3 - Viewing the result**

  This creates vendor directories in your current working directory and compiles the vendor files into them.

```
PS> dir
    Directory: D:\temp\ghdl

Mode            LastWriteTime       Length Name
----            -------------       ------ ----
d----    09.03.2018    19:33        <DIR> altera
d----    09.03.2018    19:38        <DIR> lattice
d----    09.03.2018    19:38        <DIR> osvvm
d----    09.03.2018    19:45        <DIR> uvvm
d----    09.03.2018    19:06        <DIR> xilinx-ise
d----    09.03.2018    19:40        <DIR> xilinx-vivado
```

8.6 Configuration Files

8.6.1 For Linux: *config.sh*

Please open the *config.sh* file and set the dictionary entries for the installed vendor tools to your tool's installation directories. Use an empty string "" for not installed tools.

config.sh:

```
declare -A InstallationDirectory
InstallationDirectory[AlteraQuartus]="/opt/Altera/17.1"
InstallationDirectory[LatticeDiamond]="/opt/Diamond/3.9_x64"
InstallationDirectory[OSVVM]="/home/<user>/git/GitHub/osvvm"
InstallationDirectory[UVVM]="/home/<user>/git/GitHub/uvvm_all"
InstallationDirectory[XilinxISE]="/opt/Xilinx/14.7"
InstallationDirectory[XilinxVivado]="/opt/Xilinx/Vivado/2017.4"
```

8.6.2 For Windows: *config.psm1*

Please open the *config.psm1* file and set the dictionary entries for the installed vendor tools to your tool's installation folder. Use an empty string "" for not installed tools.

config.psm1:

```
$InstallationDirectory = @{
  "AlteraQuartus" =   "C:\Altera\17.1";
  "LatticeDiamond" =  "C:\Lattice\Diamond\3.9_x64";
  "XilinxISE" =       "C:\Xilinx\14.7\ISE_DS";
  "XilinxVivado" =    "C:\Xilinx\Vivado\2017.4";
  "OSVVM" =           "D:\git\GitHub\osvvm";
  "UVVM" =            "D:\git\GitHub\uvvm_all"
}
```

8.6.3 Selectable Options for the Bash Scripts:

- Common parameters to most scripts:

```
--help, -h            Print the embedded help page(s).
--clean, -c           Cleanup directory before analyzing.
--no-warnings, -n     Don't show warnings. Report errors only.
--skip-existing, -s   Skip already compiled files (an *.o file exists).
--skip-largefiles, -S Don't compile large entities like DSP and PCIe␣
↪primitives.
--halt-on-error, -H   Stop compiling if an error occurred.
```

- *compile-altera.sh*

  Selectable libraries:

```
--all, -a             Compile all libraries, including common libraries,␣
↪packages and device libraries.
--altera              Compile base libraries like 'altera' and 'altera_mf'
--max                 Compile device libraries for Max CPLDs
--arria               Compile device libraries for Arria FPGAs
--cyclone             Compile device libraries for Cyclone FPGAs
--stratix             Compile device libraries for Stratix FPGAs
```

  Compile options:

```
--vhdl93              Compile selected libraries with VHDL-93 (default).
--vhdl2008            Compile selected libraries with VHDL-2008.
```

- *compile-xilinx-ise.sh*

  Selectable libraries:

```
--all, -a             Compile all libraries, including common libraries,␣
↪packages and device libraries.
--unisim              Compile the unisim primitives
--unimacro            Compile the unimacro macros
--simprim             Compile the simprim primitives
--corelib             Compile the xilinxcorelib macros
--secureip            Compile the secureip primitives
```

  Compile options:

```
--vhdl93              Compile selected libraries with VHDL-93 (default).
--vhdl2008            Compile selected libraries with VHDL-2008.
```

- *compile-xilinx-vivado.sh*

  Selectable libraries:

```
--all, -a             Compile all libraries, including common libraries,␣
↪packages and device libraries.
--unisim              Compile the unisim primitives
--unimacro            Compile the unimacro macros
--secureip            Compile the secureip primitives
```

  Compile options:

```
--vhdl93              Compile selected libraries with VHDL-93 (default).
--vhdl2008            Compile selected libraries with VHDL-2008.
```

- *compile-osvvm.sh*

  Selectable libraries:

```
--all, -a             Compile all.
--osvvm               Compile the OSVVM library.
```

- *compile-uvvm.sh*

  Selectable libraries:

```
--all, -a             Compile all.
--uvvm                Compile the UVVM libraries.
```

8.6.4 Selectable Options for the PowerShell Scripts:

- Common parameters to all scripts:

```
-Help                 Print the embedded help page(s).
-Clean                Cleanup directory before analyzing.
-SuppressWarnings     Don't show warnings. Report errors only.
```

- *compile-altera.ps1*

  Selectable libraries:

```
-All                  Compile all libraries, including common libraries,␣
↪packages and device libraries.
-Altera               Compile base libraries like 'altera' and 'altera_mf'
-Max                  Compile device libraries for Max CPLDs
-Arria                Compile device libraries for Arria FPGAs
-Cyclone              Compile device libraries for Cyclone FPGAs
-Stratix              Compile device libraries for Stratix FPGAs
```

  Compile options:

```
-VHDL93               Compile selected libraries with VHDL-93 (default).
-VHDL2008             Compile selected libraries with VHDL-2008.
```

- *compile-xilinx-ise.ps1*

  Selectable libraries:

```
-All                  Compile all libraries, including common libraries,␣
↪packages and device libraries.
-Unisim               Compile the unisim primitives
-Unimacro             Compile the unimacro macros
-Simprim              Compile the simprim primitives
-CoreLib              Compile the xilinxcorelib macros
-Secureip             Compile the secureip primitives
```

  Compile options:

```
-VHDL93               Compile selected libraries with VHDL-93 (default).
-VHDL2008             Compile selected libraries with VHDL-2008.
```

- *compile-xilinx-vivado.ps1*

  Selectable libraries:

```
-All                  Compile all libraries, including common libraries,␣
↪packages and device libraries.
-Unisim               Compile the unisim primitives
-Unimacro             Compile the unimacro macros
-Secureip             Compile the secureip primitives
```

  Compile options:

```
-VHDL93               Compile selected libraries with VHDL-93 (default).
-VHDL2008             Compile selected libraries with VHDL-2008.
```

- *compile-osvvm.ps1*

  Selectable libraries:

```
-All                    Compile all.
-OSVVM                  Compile the OSVVM library.
```

- *compile-uvvm.ps1*

  Selectable libraries:

```
-All                    Compile all.
-UVVM                   Compile the UVVM libraries.
```

Part II

GHDL usage

CHAPTER 9

Quick Start Guide

Since this 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. Nevertheless, multiple examples are provided, in the hope that they are useful for users to learn about both *GHDL* and *VHDL*. For advanced examples using specific features see *Examples*.

As explained in *What is GHDL?*, *GHDL* is a compiler which translates *VHDL* files to machine code. Hence, the regular workflow is composed of three steps:

- *Analysis [-a]*: convert design units (*VHDL* sources) to an internal representation.
- *Elaboration [-e]*: generate executable machine code for a target module (top-level entity).
- *Run [-r]*: execute the design to test the behaviour, generate output/waveforms, etc.

The following tips might be useful:

- Don't forget to select the version of the VHDL standard you want to use (see *VHDL standards*). The default is `--std=93c`. Use `--std=08` for VHDL-2008 (albeit not fully implemented).
  - Use `--ieee=synopsys` if your design depends on a non-standard implementation of the IEEE library.
  - Use `-fexplicit` and `-frelaxed-rules` if needed.
- Use `--work=LIB_NAME` to analyze files into the `LIB_NAME` library. To use files analyzed to a different directory, give the path to the `LIB_NAME` library using `-P/path/to/name/directory/`.
- Use the same options for analysis and elaboration. E.g., first analyse with `ghdl -a --std=08 --work=mylib myfile.vhdl`; and then elaborate and run with `ghdl --elab-run --std=08 top`.

Due to the fact that *VHDL* is processed as a general purpose language (instead of an *HDL*), all the language features are to be supported. I.e., *VHDL* sources do not need to be limited to the synthesisable subset. However, distinction between synthesisable and non-synthesisable (simulation-only) subsets is often misleading for users who are new to the language. Different examples are provided, in the hope of helping understand the different use cases:

9.1 *Hello world* program

To illustrate the general purpose of *VHDL*, the following block is a commented *Hello world* program which is saved in a file named `hello.vhdl`:

```
--  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;
```

Tip:

- Both `.vhdl` and `.vhd` extensions are used for *VHDL* source files, while `.v` is used for Verilog.
  - Since, extension `.vhd` is also interpreted as a Virtual Hard Disk file format, some users prefer `.vhdl`, to avoid ambiguity. This is the case with *GHDL*'s codebase. However, in order to maintain backward-compatibility with legacy DOS systems, other users prefer `.vhd`.
- Unless you use especial characters, either *UTF-8* or *ISO-8859-1* encodings can be used. However, if you do, the latter should be used. The standard defines ASCII (7-bit encoding) or ISO Latin-1 (ISO-8859-1) as default. However, GHDL has a relaxing option, `--mb-comments` (multi byte), to allow UTF-8 or other encodings in comments.

- First, you have to compile the file; this is called *analysis* of a design file in *VHDL* terms. Run `ghdl -a hello.vhdl` in the *shell*. This command creates or updates a file `work-obj93.cf`, which describes the library `work`.
- Then, run `ghdl -e hello_world` in the *shell*. Command `-e` means *elaborate*, which is used to build a design, with the `hello_world` entity at the top of the hierarchy.
- Last, you can directly launch the simulation *running* `ghdl -r hello_world` in the *shell*. The result of the simulation will be shown on screen:

```
Hello world!
```

Hint: If a GCC/LLVM variant of *GHDL* is used:

- *Analysis* generates a file, `hello.o`, which is the object file corresponding to your *VHDL* program. This is not created with *mcode*. These kind of object files can be compiled into foreign programs (see *Linking GHDL to Ada/C*).
- The *elaboration* step is mandatory after running the analysis and prior to launching the simulation. This will generate an executable binary named `hello_world`.
- As a result, `-r` is just a passthrough to the binary generated in the *elaboration*. Therefore, the executable can be run directly: `./hello_world`. See `-r` for more informartion.

Hint: `-e` can be bypassed with *mcode*, since `-r` actually elaborates the design and saves it on memory before running the simulation. But you can still use it to check for some elaboration problems.

9.2 *Heartbeat* module

Although *Hello world* illustrates that *VHDL* is supported as a general purpose language, the main use case of *GHDL* is to simulate hardware descriptions. The following block, which is saved in a file named `heartbeat.vhdl`, is an example of how to generate a 100 MHz clock signal with non-synthesisable VHDL:

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

entity heartbeat is
  port ( clk: out std_logic);
end heartbeat;

architecture behaviour of heartbeat
is
  constant clk_period : time := 10 ns;
begin
  -- Clock process definition
  clk_process: process
  begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
  end process;
end behaviour;
```

It can be *analysed*, *elaborated* and *run*, as you already know:

```
ghdl -a heartbeat.vhdl
ghdl -e heartbeat
ghdl -r heartbeat
```

However, execution of the design does not terminate. At the same time, no output is shown on screen. This is because, traditionally, hardware designs are continuously running devices which do not have a screen where to print. In this context, inspection and verification of the behaviour is done through waveforms, which is supported by *GHDL* (see *Export waveforms*). You can use either `--wave`, `--vcd`, `--vcdgz` or `--fst` to save the signals of the simulation to a file. Then, terminate the execution (`C-c`) and you can inspect the wave with a viewer, such as GtkWave. As explained in the manual, GtkWave *'relies on a post-mortem approach through the use of dumpfiles'*. Therefore, you should first simulate your design and dump a waveform file, say GHW:

```
ghdl -r heartbeat --wave=wave.ghw
```

Then, you can view the dump:

```
gtkwave wave.ghw
```

Of course, manually terminating the simulation is for illustration purposes only. In *Full adder* and *Working with non-trivial designs*, you will see how to write a testbench to terminate the simulation programmatically.

9.3 *Full adder* module and testbench

Unlike *Heartbeat*, the target hardware design in this example is written using the synthesisable subset of *VHDL*. It is a full adder described in a file named `adder.vhdl`:

```
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 assignments.
  --  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 *analyse* this design file, `ghdl -a adder.vhdl`, and 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. The *testbench* is a description of how to generate inputs and how to check the outputs of the Unit Under Test (UUT). This one 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. A file named `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'),
```

(continues on next page)

(continued from previous page)

```
      ('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 file, `ghdl -a adder_tb.vhdl`.

Hint: Then, if required, *elaborate* the testbench: `ghdl -e adder_tb`. You do not need to specify which object files are required, since *GHDL* knows them and automatically adds them.

Now, it is time to *run* the testbench, `ghdl -r adder_tb`, and check the result on screen:

```
adder_tb.vhdl:52:7:(assertion note): end of test
```

If your design is rather complex, you'd like to inspect signals as explained in *Heartbeat*.

See section *Simulation options*, for more details on other runtime options.

9.4 Working with non-trivial designs

Designs are usually more complex than the previous examples. Unless you are only studying VHDL, you will work with larger designs. Let's see how to analyse a 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 at ghdl.free.fr/dlx.tar.gz .

- First, untar the sources: `tar zxvf dlx.tar.gz`.

Hint:

In order not to pollute the sources with the artifacts (*WORK* library), it is a good idea to create a `work/` subdirectory. 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
```

- Then, we will run the `dlx_test_behaviour` design. We need to analyse 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`.
- GHDL knows all the design units of the DLX, but none of them has been analysed. Run the *make* command, `ghdl -m --workdir=work dlx_test_behaviour`, which analyses and elaborates a design. This creates many files in the `work/` directory, and (GCC/LLVM only) the `dlx_test_behaviour` executable in the current directory.

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

- Now, you can *run* the test suite: `ghdl -r --workdir=work dlx_test_behaviour`. The test bench monitors the bus and displays each executed instruction. It finishes with an assertion of severity level note:

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

- Last, 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 testbench in batch mode. However, you may force the simulator to stop when an assertion above or equal a certain severity level occurs. To do so, call run with this option instead: `ghdl -r --workdir=work dlx_test_behaviour --assert-level=note`. With *`--assert-level`*, the program stops just after the previous message:

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

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

- *Clean* the design library with `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 `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 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.

Warning: Sometimes, a design does not fully follow the VHDL standards. For example it might use the badly engineered `std_logic_unsigned` package. GHDL supports this VHDL dialect through some options: *`--ieee=synopsys`*, *`-fexplicit`*, etc. See section *IEEE library pitfalls*, for more details.

CHAPTER 10

Invoking GHDL

The form of the **ghdl** command is `ghdl command [options...]`. There are multiple available commands, but these general rules apply:

- 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.

Hint: 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).

Hint: Only the most common commands and options are shown here. For the most advanced and experimental features see section *Command Reference*.

Warning: 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 order of priority):

- 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 it corresponds to the value of the `HKLM\Software\Ghdl\Install_Dir` registry entry on Windows.

You should use the *--disp-config* command to display and debug installation problems.

10.1 Design building commands

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

10.1.1 Analysis [-a]

-a <[options...] file...>

Analyzes/compiles one or more files, and creates an object file for each source file. 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 (remaining files are not analyzed).

See *Synthesis command*, for details on the GHDL options. For example, to produce debugging information such as line numbers, use: `ghdl -a -g my_design.vhdl`.

10.1.2 Elaboration [-e]

-e <[options...] primary_unit [secondary_unit]>

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 object files required for the executable. Then, it links all these files with the runtime library.

- The elaboration command, *-e*, must be followed by a name of either:
  - a configuration unit
  - an entity unit
  - 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 *Synthesis command*. See section *Top entity*, for the restrictions on the root design of a hierarchy.
- If the GCC/LLVM 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 and the the filename is the name of the primary unit, or for the latter case, the concatenation of the name of the primary unit, a dash, and the name of the secondary unit (or architecture). Option `-o` followed by a filename can override the default executable filename.
- If mcode is used, this command elaborates the design but does not generate anything. Since the run command also elaborates the design, this can be skipped.

> **Warning:** This elaboration command is not a complete elaboration in terms of the VHDL standard. The actual elaboration is performed at runtime. Therefore, in order to get a complete VHDL elaboration without running the simulation, `ghdl --elab-run --no-run` is required.

10.1.3 Run [-r]

-r <[options...] primary_unit [secondary_unit] [simulation_options...]>

Runs/simulates a design. The options and arguments are the same as for the *elaboration command*.

- GGC/LLVM: simply, the filename of the executable is determined and it is executed. Options are ignored. You may also directly execute the program. The executable must be in the current directory.
- mcode: the design is elaborated and the simulation is launched. As a consequence, you must use the same options used during analysis.

This command exists for three reasons:

- You are using GCC/LLVM, but you don't need to create the executable program name.
- It is coherent with the *-a* and *-e* commands.

- It works with mcode implementation, where the executable code is generated in memory.

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

10.1.4 Elaborate and run [--elab-run]

--elab-run <[elab_options...] primary_unit [secondary_unit] [run_options...]>

Acts like the elaboration command (see -e) followed by the run command (see -r).

10.1.5 Check syntax [-s]

-s <[options] files>

Analyze files but do not generate code. This command may be used to check the syntax of files. It does not update the library.

10.1.6 Analyze and elaborate [-c]

-c <[options] file... -<e|r> primary_unit [secondary_unit]>

Hint: With GCC/LLVM, -e should be used, and -r with mcode.

The files are first parsed, and then a elaboration is performed, which drives an analysis. Effectively, analysis and elaboration are combined, but there is no explicit call to -a. With GCC/LLVM, code is generated during the elaboration. With mcode, the simulation is launched after the elaboration.

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 a smaller executable, since unused units and subprograms do not generate code.

Hint: However, you should know that most of the time is spent in code generation and the analyze and elaborate command generates 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.

Warning: This command is still under development. In case of problems, you should go back to the traditional way.

10.2 Design rebuilding commands

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

10.2.1 Import [-i]

-i <[options] file...>

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

Hint:

- Note that all the files are added to the work library. If you have many libraries, you must use the command for each library.
- Since the files are parsed, there must be correct files. However, since they are not analyzed, many errors are tolerated by this command.

See *-m*, to actually build the design.

10.2.2 Make [-m]

-m <[options] primary [secondary]>

Analyze automatically outdated files and elaborate a design. 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. A file may be outdated because it has been modified (e.g. you have just 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 option `--bind`, 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 option `-f` (force), GHDL analyzes all the units of the work library needed to create the design hierarchy. 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 files. 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.

10.2.3 Generate Makefile [--gen-makefile]

--gen-makefile <[options] primary [secondary]>

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

10.2.4 Generate dependency file command [--gen-depends]

--gen-depends <[options] primary [secondary]>

Generate a Makefile containing only dependencies to build a design unit.

This command works like the make and gen-makefile commands (see *-m*), but instead of a full makefile only dependencies without rules are generated on the standard output. Theses rules can then be integrated in another Makefile.

10.3 Synthesis command

GHDL supports synthesis, but only as a front-end: it generates a generic netlist that is not optimized.

10.3.1 Synthesis [--synth]

--synth <[options] [unit]>

--synth <[options] files -e [unit]>

The first command elaborates for synthesis the design whose top unit is indicated by `unit`. All the units must have been analyzed. The second form analyze only the files present on the command line and then elaborate them starting from the top `unit`. A generic netlist is then displayed using a very simple vhdl subset.

The command line is the same as the `ghdl` command added to Yosys by the ghdl plugin except `--synth` is not present. With this plugin, it is possible to optimize and map to a target the netlist.

This command is useful for checking that a design could be synthesized.

10.4 Options

--work=<LIB_NAME>
: Specify the name of the `WORK` library. Analyzed units are always placed in the library logically named `WORK`. With this option, you can set its name. By default, the name is `work`.

  GHDL checks whether `WORK` is a valid identifier. Although being more or less supported, the `WORK` identifier should not be an extended identifier, since the filesystem may prevent it from working correctly (due to case sensitivity or forbidden characters in filenames).

  VHDL rules forbid you from adding units to the `std` library. Furthermore, you should not put units in the `ieee` library.

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

-fsynopsys
: Allow the use of synopsys non-standard packages (`std_logic_arith`, `std_logic_signed`, `std_logic_unsigned`, `std_logic_textio`). These packages are present in the ieee library but without this option it's an error to use them.

  The synopsys 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`.

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

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

standard Supply an *IEEE* library containing only packages defined by ieee standards. Currently, there are the multivalue logic system package 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 section *VITAL packages*, for more details.

synopsys This option is now deprecated. It is equivalent to --ieee=standard and -fsynopsys.

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 section *IEEE library pitfalls*, for an example.

Warning: 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 do better to fix your design and use the numeric_std package.

-frelaxed

-frelaxed-rules
Within an object declaration, allow references to 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 starts at the *constant* keyword. Because the constant state1 and the enumeration literal state1 are homographs, 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 section *PSL support* for more details.

--format=<FORMAT>
Define the output format of some options, such as *--pp-html* or *--xref-html*.

- By default or when *--format=html2* is specified, generated files follow the HTML 2.0 standard, and colours are specified with <*FONT*> tags. However, colours are hard-coded.

- If `--format=css` is specified, generated 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.

--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 section *VHDL restrictions for VITAL* for more details.

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

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

10.5 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.

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

Hint: The warnings `-Wbinding`, `-Wlibrary`, `-Wshared`, `-Wpure`, `-Wspecs`, `-Whide`, `-Wport` are enabled by default.

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

--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 components if you don't use configuration. See section *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 still 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-reserved
Emit a warning if an identifier is a reserved word in a later VHDL standard.

--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 parentheses.

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

--warn-shared
Emit a warning when a shared variable is declared and its type it not a protected type.

--warn-hide
Emit a warning when a declaration hides a previous hide.

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

--warn-others
Emit a warning is an *others* choice is not required because all the choices have been explicitly covered.

--warn-pure
Emit a warning when a pure rules is violated (like declaring a pure function with access parameters).

--warn-static
Emit a warning when a non-static expression is used at a place where the standard requires a static expression.

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

10.6 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 the user can easily know how to disable it.

-fcaret-diagnostics

-fno-caret-diagnostics
Control whether the source line of the error is displayed with a caret indicating the column of the error.

10.7 Library commands

A new library is created implicitly, by compiling entities (packages etc.) into it: `ghdl -a --work=my_custom_lib my_file.vhdl`.

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.vhdl`. See also the *-P* command line option.

Furthermore, GHDL provides a few commands which act on a library:

10.7.1 Directory [--dir]

--dir <[options] [libs]>

Displays the content of the design libraries (by default the `work` library). All options are allowed, but only a few are meaningful: *--work*, *--workdir* and *--std*.

10.7.2 Clean [--clean]

--clean <[options]>

Try to remove any object, executable or temporary file it could have created. Source files are not removed. The library is kept.

10.7.3 Remove [--remove]

--remove <[options]>

Acts like the clean command but removes the library too. Note that after removing a design library, the files are not known anymore by GHDL.

10.7.4 Copy [--copy]

--copy <--work=name [options]>

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

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

10.8 VPI build commands

These commands simplify the compile and the link of a user vpi module. They are all wrappers: 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.

10.8.1 compile [--vpi-compile]

--vpi-compile <command>

Add an 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
```

10.8.2 link [--vpi-link]

--vpi-link <command>

Add a 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
```

10.8.3 cflags [--vpi-cflags]

--vpi-cflags

Display flags added by *--vpi-compile*.

10.8.4 ldflags [--vpi-ldflags]

--vpi-ldflags

Display flags added by *--vpi-link*.

10.8.5 include dir [--vpi-include-dir]

--vpi-include-dir

Display the include directory added by the compile flags.

10.8.6 library dir [--vpi-library-dir]

--vpi-library-dir

Display the library directory added by the link flags.

10.9 IEEE library pitfalls

When you use options *--ieee=synopsys*, 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 out, their use has pitfalls. For example, this description has an 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 (two 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 semantics of an unsigned comparison.

With some analysers, 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 (see *Synthesis command* for further 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 provide 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;
```

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

CHAPTER 11

Simulation and runtime

11.1 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.

However, the GHDL runtime behaviour can be modified with some options. For example, it is possible to pass parameters to your design through the generic interfaces of the top entity. It is also 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. For further info, see *Debugging*.

-gGENERIC=VALUE
: Set value *VALUE* to generic with name *GENERIC*.

  Example:

```
$ ghdl -r --std=08 my_unit -gDEPTH=12
```

  Warning: This is currently a run option; but in the future it will be deprecated to become an elaboration option only.

--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 from stopping simulation.

--ieee-asserts=<POLICY>
: Select how the assertions from `ieee` units are handled. *POLICY* can be `enable` (the default), `disable`

which disables all assertions from `ieee` packages and `disable-at-0` which disables only at the start of simulation.

This option can be useful to avoid assertion messages 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. The default is 5000.

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

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

--max-stack-alloc=<N>
Emit an error message in case of allocation on the stack of an object larger than *N* KB. Use 0 to disable these checks.

--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 uses the typical delay.

See section *Backannotation*, for more details.

--vpi=<FILENAME>
Load VPI module.

--vpi-trace=<FILE>
Trace vpi calls to FILE.

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

11.2 Export waveforms

--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

# Dump every signal named reset in first level sub entities of top
/top/*/reset

# Dump every signal named reset in recursive sub entities of top
/top/**/reset

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

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

# Dump every signal 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*).

--vcd-nodate
Do not write date in the VCD file.

--fst=<FILENAME>
Write the waveforms into an *fst* file 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.

11.3 Export hierarchy and references

--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
Stop the simulation before the first cycle. This may be used with *--disp-tree* to display the tree without simulating the whole design. This option actually elaborates the design, so it will catch any bound error in port maps.

--xref-html `[options] files...`
To easily navigate through your sources, you may generate cross-references. 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. An index of the files is created too.

The set of `files` 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. The style of the html file can be modified with the *--format* option.

--psl-report=<FILENAME>
Write a report for PSL at the end of simulation. For each PSL cover and assert statements, the name, source location and whether it passed or failed is reported. The file is written using the JSON format, but is still human readable.

--file-to-xml
Outputs an XML representation of the decorated syntax tree for the input file and its dependencies. It can be used for VHDL tooling using semantic information, like style checkers, documentation extraction, complexity estimation, etc.

Warning:

- The AST slightly changes from time to time (particularly when new nodes are added for new language features), so be liberal in what is allowed by your tool. Also, the XML can be quite large so consider it only during prototyping.

- Note that at this time there is no XML dump of the elaborated design.

CHAPTER 12

Interfacing to other languages

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

Hint: VHPIDIRECT is the simplest way to call C code from VHDL. VHPI is a complex API to interface C and VHDL, which allows to inspect the hierarchy, set callbacks and/or assign signals. GHDL does not support VHPI. For these kind of features, it is suggested to use VPI instead (see *VPI build commands*).

12.1 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 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 *Wrapping and starting a GHDL simulation from a foreign program*.

12.1.1 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 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, 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 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.

12.2 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 *-Wl,personal.o toplevelEntityName* as arguments. Additional object files are flagged separate *-Wl,** arguments. The elaboration step will compile the executable with the custom resources. Further reading (particularly about the backend restrictions) is at *Elaboration [-e]* and *Run [-r]*.

12.3 Wrapping and starting a GHDL 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);
```

Tip: Don't forget to list the object file of this entry point as per *Linking foreign object files to GHDL*.

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.

12.4 Linking GHDL to Ada/C

As explained previously in *Wrapping and starting a GHDL simulation from a foreign program*, 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`.

12.5 Dynamically loading foreign objects from GHDL

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";
```

12.6 Dynamically loading GHDL

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). Furthermore, when the binary is built, argument `-Wl,-pie` needs to be provided.

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
_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.

12.7 Using GRT 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.
  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 13

Command Reference

Hint: The most common commands and options are shown in section *Invoking GHDL*. Here the advanced and experimental features are described.

13.1 Environment variables

GHDL_PREFIX

13.2 Misc commands

There are a few GHDL commands which are seldom useful.

13.2.1 Help [-h]

--help, -h

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 that second command are displayed:

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

13.2.2 Display config [--disp-config]

--disp-config <[options]>

Display the program paths and options used by GHDL. This may be useful to track installation errors.

13.2.3 Display standard [--disp-standard]

--disp-standard <[options]>

Display the `std.standard` package.

13.2.4 Version [--version]

--version, -v

Display the GHDL version.

13.3 File commands

The following commands act on one or several files. These are not analyzed, therefore, they work even if a file has semantic errors.

13.3.1 Pretty print [--pp-html]

--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.

13.3.2 Find [-f]

-f <file...>

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

13.3.3 Chop [--chop]

--chop <files...>

The provided files are read, and a file is written in the current directory for every design unit. Each filename is built according to the type:

- 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 doesn't have enough memory to compile such files. The size of the executable is reduced too.

13.3.4 Lines [--lines]

--lines <files...>

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

13.4 GCC/LLVM only commands

13.4.1 Bind [--bind]

--bind <[options] primary_unit [secondary_unit]>

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

Hint: Currently, the objects generated by *--bind* are created in the working directory. This behaviour is different from other object files generated with *-a*, which are always placed in the same directory as the *WORK* library. It is possible to provide an output path with `ghdl --bind -o path/primary_unit primary_unit`. However, `ghdl --list-link` will only search in the current path.

13.4.2 Link [--link]

--link <[options] primary_unit [secondary_unit]>

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.

13.4.3 List link [--list-link]

--list-link <primary_unit [secondary_unit]>

This command may be used only after a bind command. GHDL displays all the files which will be linked to create an executable and additional arguments for the linker. This command is intended to add object files in a link of a foreign program. This command should be used only after `ghdl --bind`, as some files generated by it are looked for in the current path.

Hint: One of the arguments returned by `--list-link` is `-Wl,--version-script=PREFIX/lib/ghdl/grt.ver`, where *PREFIX* is the installation path of GHDL. This will hide most of the symbols when the target executable binary is built. In some contexts, where the binary is to be loaded dynamically, the user might want additional symbols to be accessible. There are two possible approaches to have it done:

- Filter the output of `--list-link` with e.g. `sed`.
- Provide an additional non-anonymous version script: `-Wl,-Wl,--version-script=file.ver`.

13.5 Options

--mb-comments, -C

Allow multi-bytes chars in a comment.

--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 the synthesizer rule.

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

This option is only useful during elaboration.

--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`.

13.6 Passing options to other programs

> **Warning:** These options are only available with GCC/LLVM.

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.

CHAPTER 14

Implementation of VHDL

14.1 VHDL standards

Unfortunately, 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).

Minor 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=STANDARD` option, where `STANDARD` is one of the list 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).

Multiple standards can be used in a design:

GROUP	VHDL Standard
87	87
93	93, 93c, 00, 02
08	08

Note: The standards in each group are considered compatible: you can elaborate a design mixing these standards. However, standards of different groups are not compatible.

14.2 PSL support

GHDL implements a subset of PSL.

14.2.1 PSL implementation

A PSL statement is considered a process, so it's 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:

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

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

```
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, see issue #662.

14.2.2 PSL usage

PSL annotations embedded in comments

GHDL understands embedded PSL annotations in VHDL files:

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

- 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.

Hint: As PSL annotations are embedded within comments, you must analyze your design with option `-fpsl` to enable PSL annotations:

```
ghdl -a -fpsl vhdl_design.vhdl
ghdl -e vhdl_design
```

PSL annotations (VHDL-2008 only)

Since VHDL-2008 PSL is integrated in the VHDL language. You can use PSL in a VHDL(-2008) design without embedding it in comments.

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

Hint: You have to use the `--std=08` option:

```
ghdl -a --std=08 vhdl_design.vhdl
ghdl -e --std=08 vhdl_design
```

PSL vunit files

GHDL supports vunit (Verification Unit) files.

```
vunit vunit_name (design_name)
{
  default clock is rising_edge(clk);
  assert always cnt /= 5 abort rst;
}
```

- A vunit can contain PSL and VHDL code.
- It is bound to a VHDL entity or an instance of it.
- The PSL vunit is in the same scope as the VHDL design it is bound to. You have access to all objects (ports, types, signals) of the VHDL design.

Hint: The PSL vunit file has to be analyzed/elaborated together with the VHDL design file, for example:

```
ghdl -a --std=08 vhdl_design.vhdl vunit.psl
ghdl -e --std=08 vhdl_design
```

14.3 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 does 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 architectures is analyzed). Therefore, if you delete or modify a source file of a unit analyzed, GHDL will refuse to use it.

14.4 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`* option of GHDL.

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

For details on `GROUP` values see section *VHDL standards*.

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.

14.5 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.

14.6 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`*.

CHAPTER 15

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 of right now.

15.1 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 impure).

15.2 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.

15.3 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 numbers can be used, provided no features added by later versions 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.

15.4 Negative constraint calculation

Negative constraint delay adjustments are necessary to handle negative constraints 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 16

Examples

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.

16.1 Data exchange through VHPIDIRECT

16.1.1 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.

16.1.2 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.

Part III

Development

CHAPTER 17

Synthesis

There is an experimental command (`--synth`) to generate RTL netlists (the format, VHDL or EDIF, is yet to be defined) from synthesisable code. For command `--synth` to be available, GHDL must be configured/built with option `--enable-synth` (GCC 8.1>= required, due to some new GNAT features which are only available in recent releases). Since this is a proof-of-concept, the output is mostly a dump of an internal structure for now. Therefore, it is not very useful, except for debugging.

Moreover, ghdlsynth is a complementary repository that lets GHDL to be loaded by yosys as a frontend plugin module, in order to generate bitstreams for some FPGA devices.

CHAPTER 18

Debugging

18.1 Simulation and runtime debugging options

Besides the options described in *Synthesis command*, *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. Moreover, some debugging options are also available, but not described here. The `--help` option lists all options available, including the debugging ones.

`--trace-signals`
: Display signals after each cycle.

`--trace-processes`
: Display process name before each cycle.

`--stats`
: Display run-time statistics.

`--disp-order`
: Display signals order.

`--disp-sources`
: Display sources while displaying signals.

`--disp-sig-types`
: Display signal types.

`--disp-signals-map`
: Display map bw declared signals and internal signals.

`--disp-signals-table`
: Display internal signals.

`--checks`
: Do internal checks after each process run.

`--activity``=<LEVEL>`
: Watch activity of LEVEL signals: LEVEL is `all`, `min` (default) or `none` (unsafe).

`--dump-rti`
: Dump Run Time Information (RTI).

--bootstrap
Allow `--work=std`

18.1.1 GNU Debugger (GDB)

> **Warning:** Debugging VHDL programs using *GDB* is possible only with GCC/LLVM.

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, it 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 catch a runtime error, such as indexing an array beyond its bounds. All error check subprograms call the `__ghdl_fatal` procedure. Therefore, to a 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 19

Coding Style

19.1 Ada

Ada subset: use only a simple (VHDL like) subset of Ada: no tasking, no controlled types... VHDL users should easily understand that subset. Allowed Ada95 features: the standard library, child packages. Use assertions.

We try to follow the 'GNU Coding Standards' when possible: comments before declarations, one space at the end of sentences, finish sentences with a dot. But: 2 spaces for indentation in code blocks.

No trailing spaces, no TAB (HT).

Subprograms must have a comment before to describe them, like:

```
--  Analyze the concurrent statements of PARENT.
procedure Sem_Concurrent_Statement_Chain (Parent : Iir);
```

The line before the comment must be a blank line (unless this is the first declaration). Don't repeat the comment before the subprogram body.

- For subprograms:
  1. Declare on one line when possible:

     ```
     function Translate_Static_Aggregate (Aggr : Iir) return O_Cnode
     ```

  2. If not possible, put the return on the next line:

     ```
     function Translate_Static_String (Str_Type : Iir; Str_Ident : Name_Id)
                                       return O_Cnode
     ```

  3. If not possible, put parameters and return on the next line:

     ```
     function Create_String_Literal_Var_Inner
       (Str : Iir; Element_Type : Iir; Str_Type : O_Tnode) return Var_Type
     ```

  4. If not possible, return on the next line:

     ```
     function Translate_Shortcut_Operator
       (Imp : Iir_Implicit_Function_Declaration; Left, Right : Iir)
      return O_Enode
     ```

5. If not possible, one parameter per line, just after subprogram name:

```
procedure Translate_Static_Aggregate_1 (List : in out O_Array_Aggr_List;
                                        Aggr : Iir;
                                        Info : Iir;
                                        El_Type : Iir)
```

6. If not possible, add a return after subprogram name:

```
function Translate_Predefined_TF_Array_Element
  (Op : Predefined_Boolean_Logical;
   Left, Right : Iir;
   Res_Type : Iir;
   Loc : Iir)
  return O_Enode
```

7. If not possible, ask yourself what is wrong! Shorten a name.

- Rule for the 'is': on a new line only if the declarative part is not empty:

```
procedure Translate_Assign (Target : Mnode; Expr : Iir; Target_Type :␣
↪Iir)
is
  Val : O_Enode;
begin
```

vs.

```
function Translate_Static_Range_Dir (Expr : Iir) return O_Cnode is
begin
```

If the parameter line is too long with the 'is', put in on a separate line:

```
procedure Predeclare_Scope_Type
  (Scope : in out Var_Scope_Type; Name : O_Ident) is
```

- Generic instantiation: put the generic actual part on a new line:

```
procedure Free is new Ada.Unchecked_Deallocation
  (Action_List, Action_List_Acc);
```

- For if/then statement:

1. 'then' on the same line:

```
if Get_Expr_Staticness (Decl) = Locally then
```

2. If not possible, 'then' is alone on its line aligned with the 'if':

```
if Expr = Null_Iir
  or else Get_Kind (Expr) = Iir_Kind_Overflow_Literal
then
```

3. For a multiline condition, 'or else' and 'and then' should start lines.

- 'Local' variable declaration: Do not initialize variables, constants must be declared before variables:

```
is
  N_Info : constant Iir := Get_Sub_Aggregate_Info (Info);
  Assoc  : Iir;
  Sub    : Iir;
begin
```

If the initialization expression has a side effect (such as allocation), do not use a constant.

19.2 Shell

Ubuntu uses *dash* instead of *bash* when a shell script is run. As a result, some functionalities, such as arrays like `array[1]`, are not supported. Therefore, build scripts in *dist/linux* should not use those functionalities unless they are sourced in a *bash* shell. The same applies to the scripts in *testsuite*.

19.3 Guidelines to edit the documentation

1) It's better for version control systems and diff tools to break lines at a sensible number of characters. Long lines appear as one diff. Also merging is more complex because merges are line based. Long unbreakable items may be longer (links, refs, etc.). We chose to use 160 chars.
2) Please indent all directive content by 3 spaces (not 2, and no tabs).
3) Please use * as an itemize character, since − and + are supported by docutils, but not officially supported by Sphinx.
4) Please underline all headlines with at least as many characters as the headline is long. Following the Python pattern for headlines the levels are:

```
############
************ (sometimes skipped in small documents)
============
--------------------
^^^^^^^^^^^^^^
```

5) It's not required to write

```
:samp:`code`
```

The default role for

```
``code``
```

is samp. `:samp:` is only required when you want to write italic text in code text.

```
:samp:`print 1+{variable}`
```

Now, variable becomes italic.

Please simplify all usages of `:samp:`code`` to ``code`` for readability. Here are the regular expressions for an editor like Notepad++:

- Search pattern:: `(.+?)`
- Replace pattern:: `\1`

6) Each backend has one folder and each platform/compiler has one file. Please note that page headlines are different from ToC headline:

```
.. toctree::
   :hidden:

   ToC entry <file1>
   file2
```

7) Documentation should not use "you", "we", ..., because it's not an interactive conversation or informal letter. It's like a thesis, everything is structured and formal. However, to make it more friendly to newcomers, we agree to allow informal language in the section *Quick Start Guide*.
8) Please keep errors to a minimum.

19.3.1 Guidelines to edit section 'Building'

We prefer a text block, which explains how a compilation works, what we can configure for that backend, etc. After that, we prefer a code block with e.g. bash instructions on how to compile a backend. A list of instructions with embedded bash lines is not helpful. An experienced, as well as novice user, would like to copy a set of instructions into the shell. But it should be stated what these instructions will do. Complex flows like for GCC, can be split into multiple shell code blocks. Moreover, we find it essential to demonstrate when and where to change directories.

We would like to see a list like:

- gcc (Gnu Compiler Collection)
- gcc-gnat (Ada compiler for GCC)
- llvm-del (LLVM development package)
- ...

The goal is also to explain what a user is installing and what the few lines in the build description do. Now they know the name, can search for similar names if they have another package manager or distro or can ask Google/Wikipedia. We often find many build receipts with cryptic shell code and to execute this unknown stuff with sudo is not comfortable. We would like to know what it does before hitting enter.

19.4 Documentation configuration

- Python snippet for Sphinx's *conf.py* to extract the current version number from Git (latest tag name). [#200, #221]
- Reference `genindex.html` from the navigation bar. [#200]
- Create "parts" (LaTeX terminology / chapter headlines) in navigation bar. [#200]
- **Intersphinx files [#200]**
  - To decompress the inventory file: *curl -s http://ghdl.readthedocs.io/en/latest/objects.inv | tail -n+5 | openssl zlib -d*. From how-to-uncompress-zlib-data-in-unix.
  - External ref and link to section:

```
:ref:`GHDL Roadmap <ghdl:CHANGE:Roadmap>`
```

  - External ref to option (no link):

```
:ghdl:option:`--ieee`
:option:`ghdl:--ieee`
```

CHAPTER 20

Roadmap | Future Improvements

We have several axes for *GHDL* improvements:

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

20.1 Documentation

- Development/Synthesis. Synthesis, ghdlsynth-beta, formal verification, etc. Copy the 'Usage' section from ghdlsynth's README (https://github.com/1138-4EB/ghdlsynth-beta#usage).
- Development/libghdl. How to interact with GHDL through `libghdl` and/or `libghdl-py`.
- Development/Related Projects. Brief discussion about similarities/differences with other open source projects such as rust_hdl or pyVHDLParser.
- Usage/Docker. Probably copy/convert `README.md` and `USE_CASES.md` in ghdl/docker #166.
- Usage/Language Server.
- Usage/Examples/Coverage. Code coverage in GHDL is a side effect of using GCC as a backend. In the future, GCC backend support might be dropped in favour of mcode and LLVM. To do so, code coverage with LLVM should be supported first. Anyway, comments/bits of info should be gathered somewhere in the docs, along with references to gcov, lcov, etc.
- Usage/Examples/UART. Dossmatik's UART and unisim guides. We have `*.doc` sources to be converted to Sphinx.
- Usage/Examples/Free Range VHDL. https://github.com/fabriziotappero/Free-Range-VHDL-book
- It is possible to add waveforms with wavedrom, since there is a sphinx extension available.

20.2 GSOC Ideas

This page contains ideas for enhancing GHDL that can fit internship programs, such as Google Summer of Code.

20.2.1 VHDL frontend for Yosys

Yosys is an open-source synthesis tool with built-in Verilog support and partial SystemVerilog support. ghdlsynth-beta is an experimental plugin for Yosys that allows to use GHDL (precisely, libghdl) as a frontend for Yosys. Although functional, *Synthesis* is work in progress: multiple features are not supported yet, and others need to be tested for bugs.

Note:

- FOSSI GSOC 2019 | VHDL front-end for Yosys
- FOSSI GSOC 2018 | VHDL Frontend for Yosys

20.2.2 Profiling support

Currently, GHDL does not include profiling features, which would allow to speed-up simulations and/or to detect hotspots in user designs.

Note:

- FOSSI GSOC 2018 | Profiling support
- #60

20.2.3 Improve LLVM backend

There are several possible enhancements to the current implementation of *LLVM backend*

- Debugging is supported with LLVM 3.5 only, although up to version 9.0 is supported for simulation.
- The C++ API of LLVM should be used instead of the C API.
- There was no real try to find the best order of optimization passes. This can significantly improve performance, since GHDL is currently single-threaded and CPU-bound.
- Code coverage is not supported.

Note:

- FOSSI GSOC 2018 | GHDL: Improve LLVM backend
- #866, #744, #286

20.2.4 Support 64-bit with mcode on Windows

The built-in in-memory code generator (*mcode backend*), is supported on 64 bit GNU/Linux, but not on Windows 64 bit. Compared to other backends, this would provide a lightweight and fast analyser, although it doesn't try to optimise.

Note:

- FOSSI GSOC 2018 | Support 64-bit with mcode on Windows
- #657

20.2.5 Mixed-language (VHDL-Verilog)

Multiple proofs of concept exist for co-execution of HDL simulators with other tools, such as QEMU. However, there is no open-source solution that allows to co-simulate VHDL and Verilog sources using recent versions of the standards. Some possible approaches for this task are:

- Use procedural interfaces, VPI or VHPIDIRECT (see *Interfacing to other languages*).
- Transpile/convert the HDLs into a common intermediate representation.
- Have GHDL use the API of another tool or the other way round.

Note:

- FOSSI GSOC 2018 | Framework for Mixed-Language Simulation
- FOSSI GSOC 2017 | Open Source Mixed-Language HDL Simulation
- #908, #800

20.2.6 Mixed-signal (Digital-Analog)

Thre are three different approaches for mixed-signal simulation with GHDL:

- Built-in VHDL-AMS support. It is currently possible to analyze VHDL-AMS files with GHDL (almost all the features are handled). However, it is analysis only (yet). A DAE solver needs to be pluged into GHDL compute the simulation.
- Co-execution of GHDL and an analog simulator through VPI or VHPIDIRECT (see *Interfacing to other languages*).
- Generation of simulation models from VHDL-AMS, like ADMS.

Note:

- #1052, #162

20.2.7 C APIs

Currently, GHDL can be wrapped in a foreign language (such as Ada or C) through VHPIDIRECT (see *Interfacing to other languages*). However, runtime management of the simulation is not supported. The API should be enhanced to support stepped execution. Moreover, interfacing with some types is not straightforward. Header files with the definition of those types would simplify data transference between language domains during simulation.

Note:

- #1059, #1053, #894, #819, #803, #800

20.2.8 Language server

ghdl-language-server is an experimental LSP server written in Python (which uses libghdl-py), along with clients for different editors (e.g. VSCode, Emacs or Vim). Although functional, it is work in progress: multiple features are not supported yet, and others need to be tested for bugs.

20.2.9 Project configuration file format

ghdl-language-server supports a configuration file named *hdl-prj.json*. The format of this file is undocumented and lightly defined. This is because it would be desirable to use a configuration format that can be shared with other similar tools, such as rust_hdl or pyVHDLParser. In the context of GHDL, the same configuration file might be used for the language server, simulation, synthesis, etc.

Note:

- ghdl/ghdl-language-server#12, jeremiah-c-leary/vhdl-style-guide#312

20.2.10 Packaging for Windows and/or macOS

GHDL can be installed with the most known package managers on GNU/Linux distributions (*apt*, *dnf*, *pacman*, etc.). However, this is not the case on Windows and/or macOS.

On Windows, *PKGBUILD* files for MSYS2 are available, but not upstreamed. Nonetheless, it would be desirable to distribute an standalone package that does not depend on a ful MSYS2 installation (see *Building GHDL from Sources*).

On macOS, a Homebrew formula might be written.

Ideally, these packages would be built/generated and tested in a CI workflow.

Note:

- msys2/MINGW-packages#5757
- #744, Homebrew/homebrew-cask#47256

Part IV

Internals

CHAPTER 21

Overview

GHDL is architectured like a traditionnal compiler. It has:

- a driver (sources in `src/ghdldrv`) to call the programs (compiler, assembler, linker) if needed.
- a library (sources in `src/grt`) to help execution at run-time.
- a front-end (sources in `src/vhdl`) to parse and analyse VHDL.
- a back-end (in fact many, sources are in `src/ortho`) to generate code.

The architecture is modular. For example, it is possible to use the front-end in the *libghdl* library for the language server or to do synthesis (sources in `src/synth`) instead of code generation.

The main work is performed by the front-end, which is documented in the next chapter.

CHAPTER 22

Front-end

Input files (or source files) are read by *files_map.ad[sb]*. Only regular files can be read, because they are read entirely before being scanned. This simplifies the scanner, but this also allows to have a unique index for each character in any file. Therefore the source location is a simple 32-bit integer whose type is *Location_Type*. From the location, *files_map* can deduce the source file (type is *Source_File_Entry*) and then the offset in the source file. There is a line table for each source file in order to speed-up the conversion from file offset to line number and column number.

The scanner (file `vhdl-scanner.ad[sb]`) reads the source files and creates token from them. The tokens are defined in file `vhdl-tokens.ads`. Tokens are scanned one by one, so the scanner doesn't keep in memory the previous token. Integer or floating point numbers are special tokens because beside the token itself there is also a variable for the value of the number.

For identifiers there is a table containing all identifiers. This is implemented by file `name_table.ad[sb]`. Each identifier is associated to a 32-bit number (they are internalized). So the number is used to reference an identifier. About one thousand identifiers are predefined (by `std_names.ad[sb]`). Most of them are reserved identifiers (or keywords). When the scanner find an identifier, it checks if it is a keyword. In that case it changes the token to the keyword token.

The procedure *scan* is called to get the next token. The location of the token and the location after the token are available to store it in the parser tree.

The main clieant of the scanner is the parser.

CHAPTER 23

AST

23.1 Introduction

The AST is the main data structure of the front-end and is created by the parser.

AST stands for Abstract Syntax Tree.

This is a tree because it is a graph with nodes and links between nodes. As the graph is acyclic and each node but the root has only one parent (the link that point to it). In the front-end there is only one root which represent the set of libraries.

The tree is a syntax tree because it follows the grammar of the VHDL language: there is for example a node per operation (like *or*, *and* or +), a node per declaration, a node per statement, a node per design unit (like entity or architecture). The front-end needs to represent the source file using the grammar because most of the VHDL rules are defined according to the grammar.

Finally, the tree is abstract because it is an abstraction of the source file. Comments and layout aren't kept in the syntax tree. Furthermore, if you rename a declaration or change the value of a literal, the tree will have exactly the same shape.

But we can also say that the tree is neither abstract, nor syntaxic and nor a tree.

It is not abstract because it contains all the information from the source file (except comments) are available in the AST, inclusing the location. So the source file can be reprinted (the name unparsed is also used) from the AST. If a mechanism is also added to deal with comments, the source file can even be pretty-printed from the AST.

It is not purely syntaxic because the semantic analysis pass decorate the tree with semantic information. For example the type of each expression and sub-expression is computed. This is necessary to detect some semantic error like assigning an array to an integer.

Finally, it is not anymore a tree because new links are added during semantic analysis. Simple names are linked to their declaration.

23.2 The AST in GHDL

The GHDL AST is described in file `vhdl-nodes.ads`.

An interesting particularity about the AST is the presence of a meta-model.

The meta-model is not formally described. What would be the meta-meta-model is very simple: there are elements and attributes. An element is composed of attributes, and an attribute is either a value (a flag, an integer, an enumeration) or a link to an element.

(When someone wants to be clever, he often speaks about meta-model in order to confuse you. Don't let him impress you. The trick is to answer him with any sentence containing 'meta-meta-model').

In the GHDL meta-mode, there are only 3 elements:

- variable list of nodes (*List*). These are like vectors as the length can be changed.
- Fixed lists of nodes (*Flist*). The length of a fixed list is defined at creation.
- Nodes. A node has a kind (*Iir_Kind* which is also defined in the file), and fields. The kind is set at creation and cannot be changed, while fields can be.

Or without using the word meta-model, the AST is composed of nodes and lists.

The meta-model describes the type of the attributes: most of them are either a node reference, a boolean flag or a enumerated type (like *Iir_Staticness*). But there are also links: a reference to another node or to a list.

The accessors for the node are generated automatically by the python script `src/xtools/pnodes.py`.

23.3 Why a meta-model ?

All ASTs could have a meta-model, because the definition of elements and attributes is very generic. But there is a detail: the definition of an element is static. So for each node, the list of attribute and their type is static and each list is a list of the same element type. So there is no bag, nor dynamic typing. This is per the definition of the meta-meta-model.

But in GHDL there is an API at the meta-model level in file `vhdl-nodes_meta.ads`. There is the list of all attribute types in enumeration *Types_Enum*. There is the list of all possible attributes in enumeration *Fields_Enum*. For a particular kind of node, you can get the list of fields with *Get_Field* and for every type, there is API to get or set any field of any node.

Having a meta-model API allows to build algorithm that deals with any node. The dumper (in file `vhdl-disp_tree.ad[sb]`) is used to dump a node and possibly its sub-nodes. This is very useful while debugging GHDL. It is written using the meta-model, so it knows how to display a boolean and the various other enumerated types, and how to display a list. To display a node, it just gets the kind of the type, prints the kind name and queries all the fields of the node. There is nothing particular to a specific kind, so you don't need to modify the dumper if you add a node.

The dumper won't be a strong enough reason by itself to have a meta-model. But the pass to create instances is a good one. When a vhdl-2008 package is instantiated, at least the package declaration is created in the AST (this is needed because there are possibly new types). And creating an instance using the meta-model is much simpler (and much more generic) that creating the instance using directly the nodes. The code to create instances is in files `vhdl-sem_inst.ad[sb]`.

The meta-model API is moslty automatically generated by the python script.

23.4 Dealing with ownership

The meta-model also structures the tree, because there is a notion of ownership: every element (but the root) has only one parent that owns it, and there are no cycle in the ownership. So the tree is really a tree.

That simplifies algorithms because it is easier to walk a tree than a graph. It is also easier to free a sub-tree than a sub-graph.

Getting a real tree from the parser might look obvious, but it is not. Consider the following VHDL declaration:

```
variable v1, v2 : std_logic_vector (1 downto 0) := "00";
```

Both variables `v1` and `v2` share the same type and the same initial value. The GHDL AST uses two different strategies:

- For the type, there is two fields in the node: `subtype_indication` and `type`. The `subtype_indication` is owned and set only on the first variable to the output of the parser. The `type` field is a reference and set on all variables to the result of analysis of `subtype_indication`.
- For the initial value, there is only one field `default_value` that is set on all variables. But the ownership is controlled by a flag in the node (an attribute) named `is_ref`. It is set to false on the first variable and true for the others.

The notion of ownership is highlighten by the Rust language, and indeed this is an important notion. The implementation of the Rust AST has to be investigated.

23.5 Node Type

TBC: 32-bit, extensions.

Part V

Index

CHAPTER 24

Index

Index

Symbols

Numbers

C

D

E

F

G

I

M

O

S

V

W