Contents

1 Introduction .................................................. 2
  1.1 Contribute ............................................... 2
  1.2 Support .................................................. 2
  1.3 Copyright notice ........................................... 3
  1.4 License .................................................. 3

2 Features ..................................................... 4
  2.1 Supported Tools ........................................... 4
  2.2 Supported Operating Systems .............................. 5
  2.3 Supported Python Version ................................. 5

3 Installing hdlmake ........................................... 6
  3.1 Linux deployment .......................................... 6
  3.2 Windows specific guidelines .............................. 7

4 Learn by example ............................................ 9
  4.1 Overview .................................................. 9
  4.2 The simplest hdlmake module ............................. 10
  4.3 A basic testbench ......................................... 10
  4.4 Running a simulation ...................................... 11
  4.5 Constraining a design for synthesis ..................... 12
  4.6 Synthesizing a bitstream .................................. 13
  4.7 Handling remote modules .................................. 16
  4.8 Custom Makefile commands ................................ 17
  4.9 Custom variables and conditional execution .......... 19
  4.10 Advanced examples ...................................... 20

5 hdlmake supported actions/commands ......................... 28
  5.1 Makefile generation (makefile) ......................... 28
  5.2 Fetching submodules for a top module (fetch) ........ 29
  5.3 Cleaning the fetched repositories (clean) ............. 29
  5.4 List modules (list-mods) .................................. 29
  5.5 List files (list-files) ...................................... 30
  5.6 Print manifest file variables description (manifest-help) ........................................... 31

6 Manifest variables description .................................. 32
  6.1 Top Manifest variables .................................... 32
6.2 Universal variables .................................................. 32
6.3 Simulation variables ................................................ 32
6.4 Synthesis variables ................................................... 33

7 Optional arguments for hd1make ........................................ 34
7.1 -h, --help ............................................................ 34
7.2 -v, --version ......................................................... 34
7.3 -a, --all .............................................................. 34
7.4 --log LOG ............................................................. 34
7.5 --logfile LOGFILE ................................................ 35
7.6 -p, --prefix ARBITRARY_CODE ................................. 35
7.7 -s, --sufix ARBITRARY_CODE .................................. 35
• genindex
• modindex
• search
CHAPTER 1

Introduction

A tool designed to help FPGA designers to manage and share their HDL code by automatically finding file dependencies, writing synthesis & simulation Makefiles, and fetching IP-Core libraries from remote repositories.

1.1 Contribute

• Wiki Pages: https://ohwr.org/projects/hdl-make/wiki
• Issue Tracker: https://ohwr.org/project/hdl-make/issues
• Source Code: https://ohwr.org/project/hdl-make

1.2 Support

If you are experiencing any issues, please let us know. You can find a dedicated support forum located at:

• https://forums.ohwr.org/c/hdl-make

If you are seeking for consultancy and training services on advanced hdlmake use cases, you can get commercial support from GL Research, the company on charge of maintaining and developing the tool.

• Company web site: https://gl-research.com
• Project manager: Javier Garcia Lasheras <jgarcia@gl-research.com>
1.3 Copyright notice

CERN, the European Organization for Nuclear Research, is the first and sole owner of all copyright of both this document and the associated source code deliverables.

1.4 License

This document is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit: http://creativecommons.org/licenses/by-sa/4.0/deed.en_US

The source code for the hdlmake project is licensed under the GPL license version 3 or later. To get more info about this license, visit the following link: http://www.gnu.org/copyleft/gpl.html
CHAPTER 2

Features

- Synthesis Makefile generation
- Simulation Makefile generation
- HDL parser and dependency solver
- GIT/SVN Support
- Multiple HDL Languages
- Multiple Tools
- Multiple Operating Systems

2.1 Supported Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Synthesis</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx ISE</td>
<td>Yes</td>
<td>n.a.</td>
</tr>
<tr>
<td>Xilinx PlanAhead</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Xilinx Vivado</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Altera Quartus</td>
<td>Yes</td>
<td>n.a.</td>
</tr>
<tr>
<td>Microsemi (Actel) Libero</td>
<td>Yes</td>
<td>n.a.</td>
</tr>
<tr>
<td>Lattice Semi. Diamond</td>
<td>Yes</td>
<td>n.a.</td>
</tr>
<tr>
<td>Xilinx ISim</td>
<td>n.a.</td>
<td>Yes</td>
</tr>
<tr>
<td>Mentor Graphics Modelsim</td>
<td>n.a.</td>
<td>Yes</td>
</tr>
<tr>
<td>Aldec Active-HDL</td>
<td>n.a.</td>
<td>Yes</td>
</tr>
<tr>
<td>Project IceStorm</td>
<td>Yes</td>
<td>n.a.</td>
</tr>
<tr>
<td>Icarus Verilog</td>
<td>n.a.</td>
<td>Yes</td>
</tr>
<tr>
<td>GHDL</td>
<td>n.a.</td>
<td>VHDL</td>
</tr>
</tbody>
</table>
2.2 Supported Operating Systems

hdlmake is supported in both 32 and 64 bits operating systems.

From version 3.0 onwards, hdlmake supports native Windows shells too, so you don’t need to cheat the system by using Cygwin like environments.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>tested on Ubuntu Precise/Trusty, CentOS 6/7</td>
</tr>
<tr>
<td>Windows</td>
<td>tested on Windows 7/8/8.1/10 CMD and PowerShell</td>
</tr>
</tbody>
</table>

2.3 Supported Python Version

<table>
<thead>
<tr>
<th>Version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python 2</td>
<td>Runs on 2.7.x</td>
</tr>
<tr>
<td>Python 3</td>
<td>Runs on 3.x</td>
</tr>
</tbody>
</table>
Installing hdlmake

3.1 Linux deployment

hdlmake is a Python application and is distributed as an easy to build and deploy source code Python package. In order to run hdlmake as a shell command, the next process has to be followed.

As a prerequisite, you must have the following programs installed in your host machine:

- **python**: you need a compatible Python deployment
- **git**: you need git for both fetching the hdlmake code and accessing to remote HDL repositories.
- **svn**: svn will only be used when accessing to remote SVN HDL repositories.

Note: In order to support Python 2.7.x and 3.x with a single codebase, the six Python package is now required to run hdlmake 3.0 version.

There are two methods to obtain and install hdlmake: via git repository or PyPI. PyPI is the preferred method if you are only interested in releases. Git may be preferred if you are a developer, or would like to stay in sync with active development.

3.1.1 PyPI and pip

Installing hdlmake via PyPI is straightforward assuming that you have pip already installed in your local Python environment.

Note: For directions on installing pip see the pip documentation.

To install directly from PyPI simply run:
pip install hdlmake

Alternatively, if you have already downloaded a source distribution, you can install it as follows:

```
pip install hdlmake-X.X.tar.gz
```

At this point, hdlmake is now installed into your active Python environment and can be run simply by executing hdlmake in your shell.

### 3.1.2 Git

Fetch the code from the official hdlmake git repository, that can be found at the next link: - http://www.ohwr.org/projects/hdl-make/repository

Once you have a valid hdlmake source tree, you can install hdlmake into your Python site-packages directly via setup.py install:

```
cd /path_to_hdlmake_sources/hdl-make
python2.7 setup.py install
```

hdlmake is now installed into your active Python environment and can be run simply by executing hdlmake in your shell.

As a developer, you may wish to avoid installing hdlmake directly into your site-packages, but can instead link directly to the sources instead:

```
cd /path_to_hdlmake_sources/hdl-make
python2.7 setup.py develop
```

Alternatively, you may choose to forgo installing anything in your Python environment and simply directly run from the source by creating a launch script. Create a launch script in /usr/bin or any other available location at shell $PATH. You can name the script as you prefer so, by doing this, multiple hdlmake versions can easily be used in the same machine. In any case, in this documentation we will consider that the name for this launch script is just hdlmake.

```
#!/usr/bin/env bash
PYTHONPATH=/path_to_hdlmake_sources/hdl-make python2.7 -m hdlmake $@
```

Once the launch script has been created, the appropriate execution rights must be set:

```
chmod +x /usr/bin/hdlmake
```

In the above examples the following nomenclature is used:

- **python2.7** is the executable of the Python deployment we want to use with hdlmake.
- **path_to_hdlmake_sources** is the absolute path in which the hdlmake source code has been fetched.
- **hdl-make** is the name of the folder created when you checked out the repo.
- **hdlmake** is the actual hdlmake package (this is not binary or a file, this is folder name).

### 3.2 Windows specific guidelines

From the new 3.0 version onwards, hdlmake supports execution on native Windows shell, including both the old cmd and the new PowerShell. In this section, you’ll find instructions on how to install and configure the tool and the required versions of the programs (Git and Make).
3.2.1 Make

Install GNU Make Win32:

- https://sourceforge.net/projects/gnuwin32/files/make/3.81

Then, we need to add to PATH system variable the Make bin folder, e.g.:

```
c:\Program Files (x86)\GnuWin32\bin
```

3.2.2 Git

Install Git-scm for Windows. If the bin dir is not added to the PATH, you’ll need it to update the system environmental variable.

- https://git-scm.com/download/win

3.2.3 Python

Install Python (2.7 or 3.x) for Windows:

- https://www.python.org/downloads/windows/

To make it available on the command line, add this to PATH (e.g. for Python 2.7):

```
c:\Python27
```

Before running hdlmake 3.0, you’ll need to install six package to work with Hdlmake (six is required to support Python 2.7 and 3.x with a single code base).

We can install six by just using the pip tool that comes with the Python deployment:

```
pip install six
```

3.2.4 Install hdlmake package

Install hdlmake using the Python installation mechanism:

```
python setup.py install
```

And be sure the following directory is in the PATH, as it will contain hdlmake.exe

```
c:\Python27\scripts
```
As a companion of `hdlmake`, we can find a folder containing some easy design examples that can serve us as both tests and design templates. This folder is named `hdl-make/tests/` and is automatically downloaded when the ```hdlmake` git repository is fetched.

## 4.1 Overview

Inside the `tests` folder, you’ll find a project called `counter`. This project has been specifically designed to serve as an easy template/test for the following features:

- Testbench simulation
- Bitstream synthesis
- Verilog/VHDL support

The first level of the `counter` directory structure is the following:

```bash
user@host:~$ tree -d -L 1 counter/
counter/
|-- modules
| `-- sim
| `-- syn
|     `-- testbench
|         `-- top
```

where each folder has the following role:

- `modules` contains the code of the design, a very simple 8-bit counter.
- `sim` contain a set of top manifests targeted to simulation by using different tools.
- `syn` contain a set of top manifests targeted to synthesis by using different tools.
- `testbench` contains a testbench for the design, covering the 8-bit counter.
- top contains a top module wrapper attaching the counter design to the pushbuttons & LEDs of a real FPGA design.

For each simulation or synthesis that can be executed, we have both Verilog and VHDL source codes for the module, testbench and top. So in every of the previous folder, we will have as children a verilog and an vhdl folder (note that ghdl only supports VHDL and iverilog only supports Verilog).

### 4.2 The simplest hdlmake module

If we take a deeper look to the modules folder we find that we really have two different hdlmake modules, one describing the counter as Verilog and other as VHDL.

```bash
user@host:~$ tree counter/modules/
counter/modules/
|-- counter
    |-- verilog
    |    |-- counter.v
    |    `-- Manifest.py
    `-- vhdl
        |-- counter.vhd
        `-- Manifest.py
```

Each of the modules contains a single file, so in the VHDL case the associated Manifest.py is just:

```python
files = [
    "counter.vhd",
]
```

While in the Verilog one the Manifest.py is:

```python
files = [
    "counter.v",
]
```

### 4.3 A basic testbench

Now, if we focus on the testbench folder, we have that we have again two modules, targeted to cover both the VHDL and the Verilog based counter modules we have just seen.

```bash
user@host:~$ tree counter/testbench/
counter/testbench/
|-- counter_tb
    |-- verilog
    |    |-- counter_tb.v
    |    `-- Manifest.py
    `-- vhdl
        |-- counter_tb.vhd
        `-- Manifest.py
```

Each of the modules contains a single testbench file written in the appropriated language, but in order to define the real project structure, the Manifest.py must include a reference to the modules under test. Thus, in the case of VHDL, the Manifest.py is:
files = [
    "counter_tb.vhd",
]
modules = {
    "local" : [ ".../..../modules/counter/vhdl" ],
}

While in Verilog the Manifest.py is:

files = [
    "counter_tb.v",
]
modules = {
    "local" : [ ".../..../modules/counter/verilog" ],
}

Note that, in both cases, the children modules are local.

4.4 Running a simulation

Now, we have all that we need to run a simulation for our simple design. If we take a look to the sim folder contents, we see that there is one folder for each of the supported simulations tools:

user@host:~$ tree -d -L 1 counter/sim
counter/sim
|-- active_hdl
|-- ghdl
|-- isim
|-- iverilog
|-- modelsim
|-- riviera
|-- vivado

As an example, let’s focus on the modelsim folder:

user@host:~$ tree counter/sim/modelsim/
counter/sim/modelsim/
|-- verilog
 | `-- Manifest.py
|-- vhdl
 | `-- Manifest.py
 | `-- vsim.do

We can see that there is a top Manifest.py for both Verilog and VHDL languages. In addition, we have a vsim.do file that contains Modelsim specific commands that are common for both HDL languages.

In the VHDL case, the top Manifest.py for Modelsim simulation is:

```plaintext
action = "simulation"
sim_tool = "modelsim"
sim_top = "counter_tb"
sim_post_cmd = "vsim -do ../vsim.do -i counter_tb"
```

(continues on next page)
modules = {
    "local" : [ "../../../testbench/counter_tb/vhdl" ],
}

And in the Verilog case, the associated Manifest.py is:

```python
action = "simulation"
sim_tool = "modelsim"
sim_top = "counter_tb"
sim_post_cmd = "vsim -do ../vsim.do -i counter_tb"
modules = {
    "local" : [ "../../../testbench/counter_tb/verilog" ],
}
```

In both cases, we can see that the `modules` parameter points to the specific VHDL or Verilog testbench, while the other fields remain the same for both of the languages.

The following common top specific Manifest variables describes the simulation:

- **action**: indicates that we are going to perform a simulation.
- **sim_tool**: indicates that modelsim is going to be the simulation we are going to use.
- **sim_top**: indicates the name of the top HDL entity/instance that is going to be simulated.
- **sim_post_cmd**: indicates a command that must be issued after the simulation process has finished.

Now, if we want to launch the simulation, we must follow the next steps. First, get into the folder containing the top Manifest.py we want to execute and run `hdlmake` without arguments. e.g. for VHDL:

```bash
user@host:~$ cd counter/sim/modelsim/vhdl
user@host:~$ hdlmake
```

This generates a simulation Makefile that can be executed by issuing the well known `make` command. When doing this, the appropriated HDL files are compiled in order following the hierarchy described in the `modules/Manifest.py` tree. Now, once the design is compiled, if we want to run an actual simulation we need to issue a specific Modelsim command:

```bash
user@host:~$ make
user@host:~$ vsim -do ../vsim.do -i counter_tb
```

But, because we have already defined a post simulation command into the Manifest.py, the generated Makefile allows us to combine the compilation and the test run in a single command. In this way, the second command is not required:

```bash
user@host:~$ make
```

If everything goes well, a graphical viewer should appear showing the simulated waveform. Note that every simulation top Manifest.py in the `sim` folder includes a tool specific `sim_post_command`, so all the simulations in this example can be generated by using the same simple command sequence that has been exposed here.

### 4.5 Constraining a design for synthesis

The `top` folder contains the a series of HDL files describing how to attach the counter design to the PushButtons & LEDs of real FPGA powered design. The set has been choses so that we have an example of every FPGA vendor
supported by the hdlmake tool.

```bash
user@host:~$ tree -d -L 1 counter/top
counter/top
|-- brevia2_dk
|-- cyclone3_sk
|-- icestick
|-- proasic3_sk
|-- spec_v4
`-- zedboard
```

If we focus on the spec_v4 folder, we can see that we have the following contents:

```bash
user@host:~$ tree counter/top/spec_v4/
counter/top/spec_v4/
|-- spec_top.ucf
|-- verilog
 | |-- Manifest.py
 | `-- spec_top.v
`-- vhdl
 | `-- Manifest.py
  `-- spec_top.vhd
```

We can see that we have two different modules, one for VHDL and one for Verilog, each one containing a top module that links the counter design module to the outer world. In addition, we have a common spec_top.ucf constraints file that defines the specific FPGA pins that are connected with each HDL design port.

In this way, the VHDL Manifest.py is:

```python
files = [ "spec_top.vhd", "../spec_top.ucf" ]
modules = {
    "local" : [ "../..//modules/counter/vhdl" ],
}
```

And the Verilog one is:

```python
files = [ "spec_top.v", "../spec_top.ucf" ]
modules = {
    "local" : [ "../..//modules/counter/verilog" ],
}
```

## 4.6 Synthesizing a bitstream

Once we have a constrained design targeted to a real FPGA board, we can generate a valid bitstream configuration file that can be downloaded into the FPGA configuration memory. In order to do that, we can use hdlmake to generate a synthesis Makefile that is able to perform the complete process by running a step-by-step flow that starts with the project generation and ends with the bitstream compiling. The main advantage of this approach is that, when synthesizing complex designs, the process can be resumed if it fails or is halted and the already performed jobs don’t need to be re-launched again.

The following are the different target names a synthesis Makefile may feature. The names are taken from Xilinx ISE, a synthesis tools that goes through all of the potential synthesis stages, but will apply to all other available tools too:

- **project**: Create the synthesis tool specific project.
• **synthesize**: Synthesize the HDL source code.
• **translate**: Translate to synthesis tool format.
• **map**: Map the translated design into FPGA building blocks.
• **par**: Execute Place & Route for the selected FPGA device.
• **bitstream**: Generate the bitstream for FPGA programming.

For each of the potential synthesis targets, a Tool Command Language (TCL) file will be created as a dependency in the Makefile. These TCL files include the tool specific commands that are then sourced to the selected tool to perform the different synthesis stages. We have chosen to use TCL files as the intermediate format as this is the de-facto standard language that has been selected by the FPGA vendors.

A TCL file associated to a specific synthesis stage can be generated without sourcing it to the tool by just calling Make with the associated target name (project.tcl, synthesize.tcl, translate.tcl, map.tcl, par.tcl, bitstream.tcl). In this way, this files can be integrated into other custom development flows.

**Note:** note that we have an additional files.tcl target. This is a dependency target for the project, and includes the TCL commands that are required all of the different design files to the tool in an appropriated way.

As a quick-start for synthesis projects development, in the syn folder we can find examples of top Manifest.py targeted to perform a bitstream generation by using all of the synthesis tools supported by hdlmake:

```
user@host:~$ tree -d -L 1 counter/syn
counter/syn
  |-- brevia2 dk_diamond
  |-- cyclone3 sk_quartus
  |-- icestick icestorm
  |-- proasic3 sk_libero
  |-- spec_v4 ise
  `-- spec_v4 planahead
     `-- zedboard vivado
```

Note that we have a different tool associated to each of the different supported vendor specific FPGA boards. The only exception is the spec_v4 design, that can be synthesized by using both Xilinx ISE and Xilinx PlanAhead.

If we focus on the spec_v4 ise test case, we can see the following contents in the associated folder:

```
user@host:~$ tree -d -L 1 counter/syn/spec_v4 ise
counter/syn/spec_v4 ise/
  |-- verilog
  |  `-- Manifest.py
  `-- vhdl
     `-- Manifest.py
```

As we can see, we have a top synthesis Manifest.py for Verilog and another one for VHDL. If we take a look to the VHDL Manifest.py, we have:

```python
target = "xilinx"
action = "synthesis"
syn_device = "xc6slx45t"
syn_grade = "-3"
syn_package = "fgg484"
syn_top = "spec_top"
syn_project = "demo.xise"
```

(continues on next page)
And for the Verilog synthesis top Manifest.py:

```python
action = "synthesis"
syn_device = "xc6slx45t"
syn_grade = "-3"
syn_package = "fgg484"
syn_top = "spec_top"
syn_project = "demo.xise"
syn_tool = "ise"
modules = {
    "local" : [ "../../../top/spec_v4/verilog" ],
}
```

We can see that the only difference is that each of the top synthesis Manifest.py points to its specific Verilog/VHDL top module describing the interface for the constrained FPGA design. The other Manifest.py variables are common for both languages and they mean:

- **action**: indicates that this is a synthesis process
- **syn_device**: indicates the specific FPGA device
- **syn_family**: indicates the specific FPGA family
- **syn_grade**: indicates the specific FPGA speed grade
- **syn_package**: indicates the specific FPGA package
- **syn_top**: indicates the name of the top HDL instance/module to be synthesized.
- **syn_project**: indicates the name of the FPGA project that is going to be created.
- **syn_tool**: indicates the specific synthesis tool that is going to be used.

Now, in order to generate the bitstream for our board, we just get into the folder containing the specific top Manifest.py for synthesis and run `hdlmake` without arguments to create the desired Makefile, e.g. for VHDL:

```
user@host:~$ cd counter/syn/spec_v4_ise/vhdl
user@host:~$ hdlmake
```

So, once `hdlmake` has already generated the Makefile, issuing a simple `make` command is enough to synthesize a valid bitstream. Then, we can issue a clean target for `make` in order to erase the most of the intermediate generated stuff and even a `mrproper` one to remove everything included the generated bitstreams.

```
user@host:~$ make
user@host:~$ make clean
user@host:~$ make mrproper
```

Note that `hdlmake` and the examples included in the `counter` test have been designed in order to be regular across the different toolchains. In this way, every top Manifest.py for synthesis in the `syn` folder can be executed to build a valid bitstream by using the same command sequence we have seen in this section.

4.6. Synthesizing a bitstream
4.7 Handling remote modules

Let’s take a simple example of how hdlmake handles repositories.

First, consider that we have a project consisting on four HDL modules and one testbench. The project root folder looks like this:

```bash
user@host:~/test/proj$ tree -d
.
|-- hdl
 |  |-- module1
 |  |-- module2
 |  |-- module3
 |  |-- module4
 |  |-- module5
|-- tb
```

Supposing that all of the modules are local to the development machine, i.e. the contents of the modules are available in the local host, the Manifest.py in tb directory should look like this:

```python
modules = {
    "local": ["../module1", "../module2", "../module3", "../module4", "../module5"]
}
```

We only have the local key and an associated five elements list containing the path to the respective local folders where the modules are stored.

This case was very trivial. Let’s try now to complicate the situation a bit. Let say, module1 is stored as a local module, module2 and module3 are stored in remote SVN repositories, module4 is stored in a GIT repository, and module5 is stored in a GITSM repository. Here we have how a sample module declaration including remote repositories would look like:

```python
modules = {
    "local": "../module1",
    "svn": [
        "http://path.to.repo/module2",
        "http://path.to.repo/module3@25"
    ],
    "git": "git@github.com:user/module4.git",
    "gitsm": "git@github.com:user/module5.git"
}
```

Now we can see that the local key just has an associated path (i.e. this is a 1-element list), while we have two additional key identifiers: svn, pointing to a list of two remote SVN repositories, and git, pointing to a single remote GIT repository.

Regarding the SVN repositories, module2 is pointing to the default/head revision while module3 SVN repository is pointing to revision number 25 by appending the @25 suffix to the repository URL.

Finally, the Git repository module4 is the only entry for the GIT module list and it is pointing to the default/master branch, but we can introduce the following extra modifiers to force a specific branch, tag or commit:

- [::] | Point to a specific branch (e.g. “develop”)
# e.g.: target branch is "develop"
"git":"git@github.com:user/module4.git::develop"

* [@@] | Point to a specific tag or commit

# e.g.: target tag is "v1.0"
"git":"git@github.com:user/module4.git@v1.0"

# e.g.: target commit is "a964df3d84f84ef1f87acb300c4946d8c33e526a"
"git":"git@github.com:user/module4.git@a964df3d84f84ef1f87acb300c4946d8c33e526a"

**Note:** if the requested Git repository is declared as a git submodule too and we do not provide an extra :: or @@ modifier in the Manifest.py, hdlmake will read the desired commit id from the respective submodule declaration and will checkout this code revision right after the repository is cloned.

Finally, the GITSM is just a standard Git repository and operates in the same way. The only difference with a standard Git repository for hdlmake consists in that once a GITSM module has been cloned, a recursive `git submodule init` and `git submodule update` process will be launched for this repository.

Now, if we run the `hdlmake fetch` command from inside the folder where the top Manifest.py is stored, hdlmake will read the local, SVN and GIT module lists and will automatically clone/fetch the remote SVN and GIT repositories. The only issue is that the modules would be fetched to the directory in which we are placed, which is not very elegant. To make the process more flexible, we can add the `fetchto` option to the manifest in order to point to the actual folder in which we want to store our remotely hosted modules.

```
fetchto = "../../ip_cores"
```

## 4.8 Custom Makefile commands

As we have already seen in the simulation example, hdlmake allows for the injection of optional external shell commands that are executed just before and/or just after the selected action has been executed. By using this feature, we can automate other custom tasks in addition to the hdlmake specific ones.

If a external command has been defined in the top Manifest, this is automatically written by hdlmake into the generated Makefile. In this way, the external commands are automatically executed in order when a `make` command is issued.

### 4.8.1 Synthesis commands

Depending on the tool you are going to use, you’ll be able to run specific pre/post commands for all the available fine grained synthesis targets: `{project, synthesize, translate, map, par, bitstream}`. This is how these custom commands works:

- `syn_pre_<stage>_cmd`: this command is executed before executing the `<stage>` synthesis step.
- `syn_post_<stage>_cmd`: this command is executed after executing the `<stage>` synthesis step.
As a very simple example, we can introduce in the Manifest.py from a Xilinx ISE design a hello world! command that will be executed just before the first synthesis target (project) and a bye, bye world! that will be executed just after the last synthesis target (bitstream):

target = "xilinx"
action = "synthesis"

syn_device = "xc6slx45t"
syn_grade = "-3"
syn_package = "fgg484"
syn_top = "spec_top"
syn_project = "demo.xise"
syn_tool = "ise"

syn_pre_project_cmd = "echo hello world!"
syn_post_bitstream_cmd = "echo bye, bye world!"

modules = {
    "local" : [ "../../../top/spec_v4/verilog" ],
}

4.8.2 Simulation commands

Now, if we want to add external commands to a simulation top makefile, the following parameters must be introduced:

As a very simple example, we can introduce both extra commands in the top simulation makefile we have previously seen:

action = "simulation"
sim_tool = "modelsim"
sim_top = "counter_tb"

sim_pre_cmd = "echo This is executed just before the simulation"
sim_post_cmd = "echo This is executed just after the simulation"
4.8.3 Multiline commands

If you need to execute a more complex action from the pre/post synthesis/simulation commands, you can point to an external shell script or program. As an alternative, you can use a multiline string in order to inject multiple commands into the Makefile.

As a first option, multiple commands can be launched by splitting a single long string into one piece per command. The drawback for this approach is that the original single line is reconstructed and inserted into the Makefile, so the specific external command Makefile target include just a single entry. This is why, in the following example, semicolons are used in order to separate the sequential commands:

```makefile
syn_pre_cmd = (
    "mkdir /home/user/Workspace/test1;",
    "mkdir /home/user/Workspace/test2;",
    "mkdir /home/user/Workspace/test3;",
    "mkdir /home/user/Workspace/test4;",
    "mkdir /home/user/Workspace/test5"
)
```

A cleaner alternative, is using a multiline text in which line return and tabulation characters have been introduced in order to separate in different lines each of the commands when they are written into the Makefiles. In the following example, this approach is exemplified:

```makefile
syn_pre_cmd = (
    "mkdir /home/user/Workspace/test1\n    mkdir /home/user/Workspace/test2\n    mkdir /home/user/Workspace/test3\n    mkdir /home/user/Workspace/test4\n    mkdir /home/user/Workspace/test5"
)
```

4.9 Custom variables and conditional execution

In order to give an extra level of flexibility when defining the files and modules that are going to be used in a specific project, hdlmake allows for the introduction of custom Python variables and code in the Manifest.py hierarchy. This is a very handy feature when different synthesis or simulation configurations in complex designs should be selected from the top level Manifest when running hdlmake.

As a very simple example of how this mechanism can be used, suppose that we want to simulate a design that uses a module for which two different hardware descriptions are available, one in VHDL and one in Verilog (mixed language is a common feature of commercial simulation tools and is an under-development feature for Icarus Verilog).

For this purpose, we introduce an if clause inside a children Manifest in which the simulate_vhdl boolean variable is used to select the content of the following modules to be scanned:

```python
if simulate_vhdl:
    print("We are using the VHDL module")
```

(continues on next page)
modules = {
    "local" : [ "../../../modules/counter/vhdl" ],
}
else:
    print("We are using the Verilog module")
modules = {
    "local" : [ "../../../modules/counter/verilog" ],
}

Now, in order to define the `simulate_vhdl` variable value, we can use two different approaches. The first one is to include this as a new variable in the top Manifest.py, i.e.:

```python
action = "simulation"
sim_tool = "modelsim"
sim_top = "counter_tb"

simulate_vhdl = False
modules = {
    "local" : [ "../../../testbench/counter_tb/verilog" ],
}
```

But we can also define the variable value by injecting custom prefix or sufix Python code from the command line when `hdlmake` is executed, e.g.:

```bash
hdlmake --sufix "simulate_vhdl = False" makefile
```

## 4.10 Advanced examples

### 4.10.1 Xilinx ISE

As a non-trivial design example of a real use case of `hdlmake` with Xilinx ISE, we have chosen the **White Rabbit PTP Core** reference design the European Organization for Nuclear Research (CERN) provides for the **Simple PCIe FMC Carrier (SPEC)**. This open-hardware platform is powered by a **Xilinx Spartan-6** and is used in multiple experimental physics facilities around the world.


In the following instructions, we will see how easy is to build the bitstream from the command line (**tested on both Windows and Linux hosts**).

We start by cloning the repository and getting into the SPEC reference design (**tested with Release v4.0**):

```bash
git clone git://ohwr.org/hdl-core-lib/wr-cores.git
cd wr-cores/syn/spec_ref_design/
```

Now, the WR PTP Core requires a series of HDL libraries that are provided under `hdlmake` format in the CERN Open Hardware repository. In this example, you have the option of fetching all of the dependencies for all of the reference designs provided in the downloaded source code by using the `git submodule` mechanism, this is:

```bash
git submodule init
git submodule update
```
Alternatively, if you only want to download the design submodule dependencies the design needs, we can use the `hdlmake fetch` feature as the required remote modules are already listed in the provided `Manifest.py`. As the Git remote modules directives doesn’t point to a specific branch or commit id, `hdlmake` will only clone the listed repositories in the `fetchto` folder and then will checkout the appropriated commits by previously interrogating the git submodule mechanism.

```
hdmlake fetch
```

Once we have all the dependencies, we can run `hdlmake` to automatically generate a synthesis Makefile. Once we have done this, we can just run `Make` to automatically generate the bitstream:

```
hdmlake make
```

If you experience any problem, please perform a `make clean` before running `Make`. The synthesis should run fine after doing this – the design comes out of the box with an Xilinx *.ise project created with `hdlmake 2.1`, while in `hdlmake 3.0` the synthesis project file is created by the `Makefile`.

If you want to regenerate the ISE project by using your custom Xilinx ISE properties, you may replace the provided `Manifest.py` with the following one and edit it accordingly. By using this code, you can exactly reproduce the CERN’s team setup:

```
target = "xilinx"
action = "synthesis"
syn_device = "xc6slx45t"
syn_grade = "-3"
syn_package = "fgg484"
syn_top = "spec_wr_ref_top"
syn_project = "spec_wr_ref.xise"
syn_tool = "ise"
syn_properties = [
    ["Auto Implementation Compile Order", "false"],
    ["Manual Implementation Compile Order", "true"],
    ["Manual Implementation Compile Order", "true"],
    ["Pack I/O Registers/Latches into IOBs", "For Outputs Only"],
    ["Generate Detailed MAP Report", "true"]
]modules = { "local" : "../../top/spec_ref_design/"}
```

### 4.10.2 Xilinx Vivado

As an advanced example for Xilinx Vivado, we have chosen the Beam Position Monitor (BPM) design from the Beam Diagnostics group of the Brazilian Synchrotron Light Laboratory (LNLS) for the AMC FMC Carrier (AFC). Equiped in Xilinx Artix-7 FPGA, it allows to connect clock source to any clock input.

- BPM Design: [https://github.com/lnls-dig/bpm-gw](https://github.com/lnls-dig/bpm-gw)

In the following instructions, we will see how easy is to build the bitstream from the command line ([tested on both Windows and Linux hosts](#)).

We start by cloning the repository and getting into the AFC reference design:

```
git clone https://github.com/lnls-dig/bpm-gw.git
cd bpm-gw/hdl/syn/afc_v3/vivado/dbe_bpm2/
```
Now, the Beam Position Monitor requires a series of HDL libraries that are provided under hdlmake format in both the CERN Open Hardware repository and GitHub. In this example, you have the option of fetching all of the dependencies for all of the reference designs provided in the downloaded source code by using the `git submodule` mechanism, this is:

```
git submodule init
git submodule update
```

Alternatively, if you only want to download the design submodule dependencies the design needs, we can use the `hdlmake fetch`, but we will need to modify the `bpm-gw/hdl/Manifest.py` so that its new content is:

```
fetchto = "ip_cores"
modules = { "local": ["modules/dbe_wishbone",
                     "modules/dbe_common",
                     "modules/fabric",
                     "modules/fmc_adc_common",
                     "modules/utils",
                     "modules/pcie",
                     "platform"],
           "git": [ "git://ohwr.org/hdl-core-lib/etherbone-core.git",
                    "https://github.com/lnls-dig/general-cores.git",
                    "https://github.com/lnls-dig/dsp-cores.git" ] }
```

**Note:** In order to allow the use of IP-XACT IP Core libraries for Vivado, the `fetchto` variable in the `Manifest.py` hierarchy is not only used as the folder to store remote hdlmake modules, but it's automatically assigned as the value for the `ip_repo_paths` Vivado project property.

Once this is done, `hdlmake` will only clone the listed repositories in the `fetchto` folder and then will checkout the appropriated commits by previously interrogating the `git submodule` mechanism. In this example, doing

```
hdlmake fetch
```

Once we have all the dependencies, we can run `hdlmake` to automatically generate a synthesis Makefile. Once we have done this, we can just run `make` to automatically generate the bitstream:

```
hdlmake
make
```

This project makes use of **custom Xilinx Vivado properties** and **custom Python code**, so it is a very valuable example as a template for custom setups. Here, you can see a copy of the top `Manifest.py` contents for the BPM design:

```
target = "xilinx"
action = "synthesis"
syn_device = "xc7a200t"
syn_grade = "-2"
syn_package = "ffg1156"
syn_top = "dbe_bpm2"
syn_project = "dbe_bpm2"
syn_tool = "vivado"
syn_properties = [
    ["steps.synth_design.args.more options", "-verbose"],
    ["steps.synth_design.args.retiming", "1"],
    ["steps.synth_design.args.assert", "1"],
]
```

(continues on next page)
import os
import sys
if os.path.isfile("synthesis_descriptor_pkg.vhd"):
    files = ["synthesis_descriptor_pkg.vhd"];
else:
    sys.exit("Generate the SDB descriptor before using HDLMake (.build_synthesis_sdb.
    
        sh)");
machine_pkg = "uvx_250M";
modules = { "local" : [ "/.\.\.\.\"../\top\afc\v3\vivado\dbe\bpm2\""] };

Note: If you are generating HDL code from a Xilinx IP library, you may need to select the target language by setting the language top manifest variable to either vhdl or verilog. If the language variable is not defined, hdlmake will choose vhdl as the default HDL language.

4.10.3 Intel Quartus

In the same source code design the CERN provides for the White Rabbit PTP Core, it is also included an example for the VME FMC Carrier HPC-DDR3 (VFC-HD). The VFC-HD is an Intel Arria V based VME64x carrier for one High Pin Count (HPC) FPGA Mezzanine Card (FMC, VITA 57). It is has six SFP+ transceivers compatible with support for rad-hard GBT links, CERN Beam Synchronous Timing (BST), White Rabbit and Ethernet.

- WR PTP Core: http://www.ohwr.org/projects/wr-cores/wiki/Current_release

We start by cloning the repository and getting into the VFC-HD reference design (tested with Release v4.0):

```
git clone git://ohwr.org/hdl-core-lib/wr-cores.git
cd wr-cores/syn/vfchd_ref_design/
```

Now, the WR PTP Core requires a series of HDL libraries that are provided under hdlmake format in the CERN Open Hardware repository. In this example, you have the option of fetching all of the dependencies for all of the reference designs provided in the downloaded source code by using the git submodule mechanism, this is:

```
git submodule init
git submodule update
```
Alternatively, if you only want to download the design submodule dependencies the design needs, we can use the `hdlmake fetch` feature as the required remote modules are already listed in the provided `Manifest.py`. As the Git remote modules directives don’t point to a specific branch or commit id, `hdlmake` will only clone the listed repositories in the `fetchto` folder and then will checkout the appropriated commits by previously interrogating the git submodule mechanism.

```
hdmake fetch
```

Once we have all the dependencies, we can run `hdlmake` to automatically generate a synthesis Makefile. Once we have done this, we can just run Make to automatically generate the bitstream:

```
hdmake
make
```

It’s important to know that this design relies on versioned Intel IP-Cores, so it will only work just out of the box with the appropriated Intel Quartus version (WR PTP Core v4.0 requires Intel Quartus 16.0).

If you want to use a different Intel Quartus version, you will need to fix the IP-Core versions. This is an example patch for upgrading WR PTP Core v4.0 from Intel Quartus 16.0 to Intel Quartus 16.1:

```
diff --git a/platform/altera/wr_arria5_phy/arria5_phy16.txt b/platform/altera/wr__arria5_phy/arria5_phy16.txt
index c0db187..8040f12 100644
--- a/platform/altera/wr_arria5_phy/arria5_phy16.txt
+++ b/platform/altera/wr_arria5_phy/arria5_phy16.txt
@@ -1,6 +1,6 @@
--- megafunction wizard: %Deterministic Latency PHY v16.0%+
++ megafunction wizard: %Deterministic Latency PHY v16.1%
 -- Retrieval info: <xml version="1.0"/>
--- Retrieval info: <instance entity-name="altera_xcvr_det_latency" version="16.0" />
++ Retrieval info: <instance entity-name="altera_xcvr_det_latency" version="16.1" />
 -- Retrieval info: <generic name="device_family" value="Arria V" />
--- Retrieval info: <generic name="operation_mode" value="Duplex" />
++ Retrieval info: <generic name="lanes" value="1" />

diff --git a/platform/altera/wr_arria5_phy/arria5_phy8.txt b/platform/altera/wr__arria5_phy/arria5_phy8.txt
index 5b52c9a..2993643 100644
--- a/platform/altera/wr_arria5_phy/arria5_phy8.txt
+++ b/platform/altera/wr_arria5_phy/arria5_phy8.txt
@@ -1,6 +1,6 @@
--- megafunction wizard: %Deterministic Latency PHY v16.0%
++ megafunction wizard: %Deterministic Latency PHY v16.1%
 -- Retrieval info: <xml version="1.0"/>
--- Retrieval info: <instance entity-name="altera_xcvr_det_latency" version="16.0" />
++ Retrieval info: <instance entity-name="altera_xcvr_det_latency" version="16.1" />
 -- Retrieval info: <generic name="device_family" value="Arria V" />
--- Retrieval info: <generic name="operation_mode" value="Duplex" />
++ Retrieval info: <generic name="lanes" value="1" />

diff --git a/platform/altera/wr_arria5_phy/arria5_phy_reconf.txt b/platform/altera/wr__arria5_phy/arria5_phy_reconf.txt
index 78e0f2f..d698be2 100644
--- a/platform/altera/wr_arria5_phy/arria5_phy_reconf.txt
+++ b/platform/altera/wr_arria5_phy/arria5_phy_reconf.txt
@@ -1,6 +1,6 @@
--- megafunction wizard: %Transceiver Reconfiguration Controller v16.0%
++ megafunction wizard: %Transceiver Reconfiguration Controller v16.1%
 -- Retrieval info: <xml version="1.0"/>
--- Retrieval info: <instance entity-name="alt_xcvr_reconfig" version="16.0" />
++ Retrieval info: <instance entity-name="alt_xcvr_reconfig" version="16.0" />
```

(continues on next page)
If you want to regenerate the Quartus project by using your custom Quartus properties, you may replace the provided Manifest.py with the following one and edit it accordingly. Note that this will generate set_global_assignment statements in which the dictionary key name is the name of property and the key value its value. Other supported property keys are tag, section_id, to and from. As an example, we force the VHDL and Verilog input version and optimize the synthesis for speed:

```python
target = "altera"
action = "synthesis"

syn_family = "Arria V"
syn_device = "5agxmb1g4f"
syn_grade = "c4"
syn_package = "40"
syn_top = "vfchd_wr_ref_top"
syn_project = "vfchd_wr_ref"
syn_tool = "quartus"
syn_properties = [
    {"name": "VHDL_INPUT_VERSION", "value": "VHDL_2008"},
    {"name": "VERILOG_INPUT_VERSION", "value": "SYSTEMVERILOG_2005"},
    {"name": "optimization_technique", "value": "speed"}
]

quartus_preflow = "quartus_preflow.tcl"

files = [
    "vfchd_wr_ref.sdc",
    "quartus_preflow.tcl",
]

modules = {
    "local": ["../top/vfchd_ref_design/",
]
}
```

### 4.10.4 Mentor Modelsim

In the same sources provided by CERN for the White Rabbit PTP core, there are several complex simulation examples for Mentor Modelsim. These examples cover different features of the WR core and use mixed VHDL, Verilog and SystemVerilog co-simulation (note that Xilinx Unisim libraries are required to be compiled for Modelsim and installed too):

```bash
git clone git://ohwr.org/hdl-core-lib/wr-cores.git
cd wr-cores
git checkout wrpc-v4.0
git submodule init
git submodule update
```

From the provided simulation examples, we will use the following one. This is a complete demo that simulates a WR PTP Core receiving and processing synchronous Gigabit Ethernet frames, from the PHY interface controller to the
embedded 32-bits soft-processor:

cd testbench/wrc_core

In some version of Modelsim (e.g. Modelsim PE 10.5a), we will need to fix the declaration of the initialized variables in System Verilog testbenches. As an example, we may apply the modifications highlighted in the following diff patch:

diff --git a/testbench/wrc_core/functions.svh b/testbench/wrc_core/functions.svh
index 33abf71..8ad1c8e 100644
--- a/testbench/wrc_core/functions.svh
+++ b/testbench/wrc_core/functions.svh
@@ -48,14 +48,14 @@ semaphore txPkt = new(1);
     */
     task send_frames(WBPacketSource src, int n_packets, int ifg = 0 /*[us]*/);
     // TODO: improve the IFG: allow to make it tighter
-    int i, seed = 0,n1=0,n2=0;
-    int cur_size, dir;
+    static int seed = 0,n1=0,n2=0;
+    int i, cur_size, dir;
    EthPacket pkt, tmpl;
    EthPacket to_ext[$], to_minic[$];
-    EthPacketGenerator gen = new;
+    static EthPacketGenerator gen = new;
    int random_ifg; //us
-    int min_ifg = 1; //us
-    int max_ifg = 100;//us
+    static int min_ifg = 1; //us
+    static int max_ifg = 100;//us
+    static int min_ifg = 1; //us
+    static int max_ifg = 100;//us

    tmpl = new;
    tmpl.src = '{'h22,'h33,'h44,'h44,'h55,'h66};

Once everything is ready, we can compile the design sources. As the HDLMake SystemVerilog parser is not very accurate yet and it is not able to successfully solve the complete hierarchy, we will need to pass the -a flag to the makefile generation command so that all the files are parsed and passed to the compiler – even if we cannot relate them to the top entity by using dependency relations:

hdltmake -a makefile
make
vsim -modelsimini modelsim.ini -L unisim -do run.do -i main

Alternatively, we may provide the full path to the Unisim library (required Xilinx sim components), e.g.:

vsim -L c:\modeltech_pe_10.5a\xilinx_libs\unisim -do run.do -i main

In this point, if everything goes OK, Modelsim window will open and we will see the waveform progress for the design under test:

**Note:** When working with Verilog and SystemVerilog included files in Modelsim and derivatives, you will need to use the include_dirs parameter in Manifest.py to specify the directories in which the files to be included can be stored. It’s important to know that the +incdir+ directives will automatically be stripped from vlog_opt by hdltmake.
4.10. Advanced examples
5.1 Makefile generation (makefile)

This is the default command for hdlmake and its basic behaviour will be defined by the value of the action manifest parameter in the hierarchy Manifest.py. action can be set to simulation or synthesis, and the associated command sequence will be:

- **simulation**: generate a simulation makefile including all the files required for the defined testbench
- **synthesis**: generate a synthesis makefile including all the files required for bitstream generation

By using the `-f FILENAME`, `--filename FILENAME` optional argument for the makefile command, we can choose the name of the synthesis or simulation Makefile that will be generated by hdlmake.

In order to allow for a more agile development, we have included these shortcuts when using the hdlmake makefile command:

```
# These commands are equivalent
hdlmake makefile
hdlmake

# These commands are equivalent
hdlmake makefile -f FILENAME
hdlmake -f FILENAME
```

**Note:** in any case, it’s supposed that all the required modules have been previously fetched. Otherwise, the process will fail.
5.2 Fetching submodules for a top module (**fetch**)

Fetch and/or update remote modules listed in Manifest. It is assumed that a projects can consist of modules, that are stored in different places (locally or a repo). The same thing is about each of those modules - they can be based on other modules. Hdlmake can fetch all of them and store them in specified places. For each module one can specify a target catalog with manifest variable `fetchto`. Its value must be a name (existent or not) of a folder. The folder may be located anywhere in the filesystem. It must be then a relative path (**hdlmake** support solely relative paths).

5.3 Cleaning the fetched repositories (**clean**)

remove all modules fetched for direct and indirect children of this module

5.4 List modules (**list-mods**)

List all the modules involved in the design described by the top manifest and its children hierarchy. The output list will be printed in stdout, being each of the modules represented by a single text line made up of two entries separated by a tab. The first entry in a line associated to a module is the relative path to the specific directory where the module is stored, while the second entry is a identifier indicating the origin of the module.

---

**Note:** the hierarchy will be built considering that the Manifest.py stored in the folder from which we all launching the command is the root one.

---

If we provide the **--with-files** argument to **list-mods**, the files being directly included by a specific module will be printed just after the line pointing to this specific module. Each of the printed lines associated to a file will be composed by two entries too, the first one pointing to the relative path to the file and the second one indicating that the resource is a file.

In addition to the lines representing modules and files, **list-mods** will also print to standard output some comments providing additional information about the structure of the module hierarchy. This modules will start with a `#` sign, following in this way the usual shell convention. By providing the **--terse** argument to **list-mods**, the comment lines won’t be printed to stdout, being only printed those lines representing an actual resource (module or file).

Summing everything up, the following table compiles the different kind of resources that can be printed when listing the modules:

<table>
<thead>
<tr>
<th>Code</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>this module is stored in the local host</td>
</tr>
<tr>
<td>git</td>
<td>this is a module hosted in a GIT repository</td>
</tr>
<tr>
<td>svn</td>
<td>this is a module hosted in a SVN repository</td>
</tr>
<tr>
<td>file</td>
<td>this is a file (included by the module that has been previously printed)</td>
</tr>
</tbody>
</table>

Now, here we have some examples of the outputs we should expect when using the **list-mods** command:

```
user@host:~/.hd1-make/tests/counter/syn/spec_v4_ise/vhdl$ hdlmake list-mods --with-files
# MODULE START -> git@ohwr.org:misc/hdl-make.git
# * This is the root module
  .  local
# * This module has no files
```

(continues on next page)
5.5 List files (list-files)

List all the files that are defined inside all of the modules included in the hierarchy in the form of a string of elements separated by a DELIMITER and print this list to stdout.

The DELIMITER is defaulted to a white space, but it can be defined by using the --delimiter DELIMITER argument for the list-files command.

Note: the file list will be built considering that the Manifest.py stored in the folder from which we are launching the command is the root one.

In order to build the file list, hdlmake will parse the HDL files to find the required dependencies that a top entity needs to be successfully compiled. We can configure the name of the HDL module that will be considered as the top entity to build the required file hierarchy by using the --top TOP optional argument to the list-files command. If no top entity is defined, all of the design files will be listed.

Finally, by using the --reverse optional argument we are able to reverse the order of the listed files.
5.6 Print manifest file variables description (manifest-help)

Print manifest file variables description
6.1 Top Manifest variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>str</td>
<td>What is the action that should be taken (simulation/synthesis)</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>incl_makefiles</td>
<td>list</td>
<td>List of .mk files included in the generated makefile</td>
<td>[]</td>
</tr>
<tr>
<td>language</td>
<td>str</td>
<td>Select the default HDL language if required (verilog, vhdl)</td>
<td>&quot;vhdl&quot;</td>
</tr>
</tbody>
</table>

6.2 Universal variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetchto</td>
<td>str</td>
<td>Destination for fetched modules</td>
<td>None</td>
</tr>
<tr>
<td>modules</td>
<td>dict</td>
<td>List of local modules</td>
<td>{}</td>
</tr>
<tr>
<td>files</td>
<td>str,</td>
<td>List of files from the current module</td>
<td>[]</td>
</tr>
<tr>
<td>library</td>
<td>str</td>
<td>Destination library for module’s VHDL files</td>
<td>work</td>
</tr>
<tr>
<td>include_dirs</td>
<td>list, str</td>
<td>Include dirs for Verilog sources</td>
<td>None</td>
</tr>
<tr>
<td>extra_modules</td>
<td>list</td>
<td>Force the listed HDL entities to be included in the design</td>
<td>None</td>
</tr>
</tbody>
</table>

6.3 Simulation variables

Basic simulation variables:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_top</td>
<td>str</td>
<td>Top level module for simulation</td>
<td>None</td>
</tr>
<tr>
<td>sim_tool</td>
<td>str</td>
<td>Simulation tool to be used (e.g. isim, vsim, iverilog)</td>
<td>None</td>
</tr>
<tr>
<td>sim_pre_cmd</td>
<td>str</td>
<td>Command to be executed before simulation</td>
<td>None</td>
</tr>
<tr>
<td>sim_post_cmd</td>
<td>str</td>
<td>Command to be executed after simulation</td>
<td>None</td>
</tr>
</tbody>
</table>
Modelsim/VSIm specific variables:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsim_opt</td>
<td>str</td>
<td>Additional options for vsim</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>vcom_opt</td>
<td>str</td>
<td>Additional options for vcom</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>vlog_opt</td>
<td>str</td>
<td>Additional options for vlog</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>vmap_opt</td>
<td>str</td>
<td>Additional options for vmap</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>modelsim_ini_path</td>
<td>str</td>
<td>Directory containing a custom modelsim.ini file</td>
<td>None</td>
</tr>
</tbody>
</table>

Icarus Verilog specific variables:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>iverilog_opt</td>
<td>str</td>
<td>Additional options for iverilog</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

GHDL specific variables:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ghdl_opt</td>
<td>str</td>
<td>Additional options for ghdl</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

### 6.4 Synthesis variables

Basic synthesis variables:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>syn_top</td>
<td>str</td>
<td>Top level module for synthesis</td>
<td>None</td>
</tr>
<tr>
<td>syn_tool</td>
<td>str</td>
<td>Tool to be used in the synthesis</td>
<td>None</td>
</tr>
<tr>
<td>syn_device</td>
<td>str</td>
<td>Target FPGA device</td>
<td>None</td>
</tr>
<tr>
<td>syn_family</td>
<td>str</td>
<td>Target FPGA family</td>
<td>None</td>
</tr>
<tr>
<td>syn_grade</td>
<td>str</td>
<td>Speed grade of target FPGA</td>
<td>None</td>
</tr>
<tr>
<td>syn_package</td>
<td>str</td>
<td>Package variant of target FPGA</td>
<td>None</td>
</tr>
<tr>
<td>syn_project</td>
<td>str</td>
<td>Project file name</td>
<td>None</td>
</tr>
<tr>
<td>syn_pre_synthesize_cmd</td>
<td>str</td>
<td>Command to be executed before synthesis: synthesize</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>syn_post_synthesize_cmd</td>
<td>str</td>
<td>Command to be executed after synthesis: synthesize</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>syn_pre_translate_cmd</td>
<td>str</td>
<td>Command to be executed before synthesis: translate</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>syn_post_translate_cmd</td>
<td>str</td>
<td>Command to be executed after synthesis: translate</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>syn_pre_map_cmd</td>
<td>str</td>
<td>Command to be executed before synthesis: map</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>syn_post_map_cmd</td>
<td>str</td>
<td>Command to be executed after synthesis: map</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>syn_pre_par_cmd</td>
<td>str</td>
<td>Command to be executed before synthesis: par</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>syn_post_par_cmd</td>
<td>str</td>
<td>Command to be executed after synthesis: par</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>syn_pre_bitstream_cmd</td>
<td>str</td>
<td>Command to be executed before synthesis: bitstream</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>syn_post_bitstream_cmd</td>
<td>str</td>
<td>Command to be executed after synthesis: bitstream</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

Altera Quartus II / Prime specific variables:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>quartus_preflow</td>
<td>str</td>
<td>Quartus pre-flow script file</td>
<td>None</td>
</tr>
<tr>
<td>quartus_postmodule</td>
<td>str</td>
<td>Quartus post-module script file</td>
<td>None</td>
</tr>
<tr>
<td>quartus_postflow</td>
<td>str</td>
<td>Quartus post-flow script file</td>
<td>None</td>
</tr>
</tbody>
</table>
Optional arguments for `hdlmake`

Hdlmake can be run with several arguments. The way of using them is identical with the standard one in Linux systems. The order of the arguments is not important. Hereafter you can find each argument with a short description.

### 7.1 `-h, --help`

Shows help message that is automatically generated with Python’s `optparse` module. Gives a short description of each available option.

### 7.2 `-v, --version`

Print the version of the `hdlmake` instance on execution and quit.

### 7.3 `-a, --all`

Disable the stage in which `hdlmake` purges the files that are considered as not dependent on the top entity. In this way, by activating this flag all of the files listed by the module hierarchy will be used for the issued action.

### 7.4 `--log LOG`

Set logging level for the Python logger facility. You can choose one of the levels in the following tables, in which the associated internal logging numeric value is also included:
<table>
<thead>
<tr>
<th>Log Level</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical</td>
<td>50</td>
</tr>
<tr>
<td>error</td>
<td>40</td>
</tr>
<tr>
<td>warning</td>
<td>30</td>
</tr>
<tr>
<td>info</td>
<td>20</td>
</tr>
<tr>
<td>debug</td>
<td>10</td>
</tr>
<tr>
<td>not provided</td>
<td>0</td>
</tr>
</tbody>
</table>

### 7.5 `--logfile LOGFILE`

Use a file to store all of the log information generated by `hdlmake`. For example, if we want to list the files contained by a design while storing the log in `/var/log/hdlmake.log`, we should run the following command:

```bash
hdlmake --logfile /var/log/hdlmake.log list-files
```

### 7.6 `--prefix ARBITRARY_CODE`

Add arbitrary Python code from the command line that will be evaluated before each `Manifest.py` parse action across the hierarchy.

As an example, this command will generate the Makefile and will try to print `Hello hdlmake` before each `Manifest.py` run:

```bash
hdlmake -p "print('Hello hdlmake')" makefile
```

### 7.7 `--sufix ARBITRARY_CODE`

Add arbitrary Python code from the command line that will be evaluated after each `Manifest.py` parse action across the hierarchy.

As an example, this command will generate the Makefile but will try to print `Bye, bye hdlmake` after each `Manifest.py` run:

```bash
hdlmake -s "print('Bye, bye hdlmake')" makefile
```