hdlmake Documentation

Release 2.0

Javier D. Garcia-Lasheras

February 08, 2015

Warning: The full project documentation is under development. Check this space as new content will be added in the coming days.

- *genindex*
- *modindex*
- *search*

CHAPTER 1

Introduction

1.1 Contribute

- Issue Tracker: http://www.ohwr.org/projects/hdl-make/issues
- Source Code: http://www.ohwr.org/projects/hdl-make/repository

1.2 Support

If you are having issues, please let us know. We have a mailing list located at: http://www.ohwr.org/mailing_list/show?project_id=hdl-make

1.3 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

1.4 Copyright notice

Features

- Synthesis
- Simulation
- GIT/SVN Support
- Multi Language
- Multi Tools
- Multiple Operating System Support

2.1 Supported Tools

Tool	Synthesis	Simulation
Xilinx ISE	Yes	n.a.
Xilinx PlanAhead	Yes	No
Altera Quartus	Yes	n.a.
Microsemi (Actel) Libero	Yes	n.a.
Lattice Semi. Diamond	Yes	n.a.
Xilinx ISim	Yes	n.a.
Mentor Graphics Modelsim	n.a.	Yes
Aldec Active-HDL	n.a.	Yes
Icarus Verilog	n.a.	Verilog
GHDL	n.a.	VHDL

2.2 Supported Operating Systems

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

Operating System	Comments
Linux	tested on Ubuntu Precise/Trusty, CentOS 6/7
Windows	tested on Windows 7/8/8.1 by using Cygwin

2.3 Supported Python Version

Version	Comments
Python 2	Runs on 2.7.x
Python 3	To be done, not supported yet

Installing `hdlmake`

3.1 Linux deployment

`hdlmake` is a Python application and, in order to allow an agile development and customization, is not distributed as a packaged executable file, but as a set of Python source files. In this way, there is no need to build hdlmake, as the Python code gets interpreted on the fly. 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.

Now, you need to 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 need to 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
python2.7 /path_to_hdlmake_sources/hdl-make/hdlmake/__main__.py $@
```

here:

- `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 subfolder of hdl-make (this is not binary or a file, this is folder name).

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

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

3.2 Windows specific guidelines

Despite the fact that `hdlmake` was originally designed to be used in Linux environments, the new release of the tool has been modified to be easily used in both 32 and 64 bits Windows Operating Systems inside a Cygwin deployment. In this way, you must just follow the next steps to be able to run `hdlmake`.

First, install a valid Cygwin environment for your Windows machine. I order to access to the full set of features from `hdlmake`, you must choose at least the following packages when deploying Cygwin:

- python (choose the most updated 2.7 release)
- openssh
- git-svn
- git
- curl
- make

Once you have installed your Cygwin environment, you can just get into the Cygwin console and operate as if you were inside a Linux machine for both installing and working with `hdlmake`.

Environment ———-_

When working in Linux or Windows inside Cygwin, in order to work with `hdlmake` we must assure that the tools executables that are going to be used are accessibles in the shell $PATH. This is a requirement for both simulation and synthesis

..warning:: there is another way to define the specific tools as an environmental variable, but this is buggy and fails when executing some of the actions. The $PATH way is the most easy and stable way to go!

Learn by example

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:

```
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.

```
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:

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

While in the Verilog one the Manifest.py is:

```
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.

```
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
|-- aldec
|-- ghdl
|-- isim
|-- iverilog
`-- modelsim
```

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:

```
action = "simulation"
sim_tool = "modelsim"
top_module = "counter_tb"

sim_post_cmd = "vsim -do ../vsim.do -i counter_tb"

modules = {
    "local" : [ "../../../testbench/counter_tb/vhdl" ],
}
```

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

```
action = "simulation"
sim_tool = "modelsim"
top_module = "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.
- `top_module`: indicates the name of the top HDL entity/instance that is going to be simulated.
- `sim_post_cmd`: indicates a command that can be issued after the simulation process has finnished.

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:

```
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 hierachy 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:

```
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:

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

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 chosed so that we have an example of every FPGA vendor supported by the `hdlmake` tool.

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

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

```
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:

```
files = [ "spec_top.vhd", "../spec_top.ucf" ]

modules = {
    "local" : [ "../../../modules/counter/vhdl" ],
}
```

And the Verilog one is:

```
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, 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
|-- proasic3_sk_libero
|-- spec_v4_ise
`-- spec_v4_planahead
```

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:

```
target = "xilinx"
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/vhdl" ],
}
```

And for the Verilog synthesis top Manifest.py:

```
target = "xilinx"
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 means:

- `target`: specific targeted FPGA architecture
- `action`: indicates that this is a synthesis process
- `syn_device`: indicates the specific FPGA device
- `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, e.g. for VHDL:

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

The `hdlmake` performs two independent actions in the next order:

1. Create an ISE project containing the all the files that are in the hierachy indicated by the Manifest.py tree. If there is an existing project in the folder, this will be updated accordingly.
2. Generate a synthesis Makefile which contains all the information for building the associated ISE project in order to get a valid bitstream.

So, once `hdlmake` has already generated the project and 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 but the bitstream and the project.

```
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.

4.7 Handling remote modules

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

Our project consists of 4 HDL modules and one testbench. Its directory looks like this:

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

Supposing that the testbench will use all modules, the manifest in `tb` directory should look like this:

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

This case was very trivial. Let's try now to complicate the situation a bit. Let say, that two of our modules are stored in a SVN repository and the last one in a GIT repository. What is more, for module2 we would like to use revision number 25. In that case, the manifest will look as follows:

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

The generated makefile will work fine. The only issue is that the modules will be fetched to the directory of testbench, which is not very elegant. To make it better, add `fetchto` to the manifest:

```
fetchto = ".."
```

This will tell Hdlmake to fetch modules to the project catalog. Let's see how it works:

```
user@host:~/test/proj$ tree -d
.
`-- hdl
    |-- module1
    `-- tb
user@host:~/test/proj$ cd hdl/tb
user@host:~/test/proj/hdl/tb$ hdlmake.py -f
user@host:~/test/proj$ cd ../..
user@host:~/test/proj$ tree -d
```

```
.
`-- hdl
    |-- module1
    |-- module2
    |-- module3
    |-- module4
    `-- tb
```

And we finally get the original project we started with.

4.8 Advanced examples

EVO project: PlanAhead synthesis project for the Zedboard platform, powered by Xilinx Zynq based ARM Dual Cortex-A9 processor plus Artix grade FPGA and performing an asynchronous logic demo: http://www.ohwr.org/projects/evo/repository

UMV, Mentor Questa & System Verilog simulation: Work in progress, ready to be merged into Main branch.

hdlmake supported actions/commands

5.1 Check environment (`check-env`)

Check environment for HDLMAKE-related settings. This scan the top Manifest and report if the potentially used tools or/and environment variables are met or not.

5.2 Print manifest file variables description (`manifest-help`)

Print manifest file variables description

5.3 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).

5.4 Cleaning the fetched repositories (`clean`)

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

5.5 List modules (`list-mods`)

List all modules involved in the design described by the top manifest. In addition to the module path & name, a code number indicating the module origin will be returned for each of the modules. These number means:

Code	Origin
1	GIT
2	SVN
3	Local

5.6 List files (`list-files`)

List all the files that are defined inside all the modules in the hierachy in the form of a space-separated string

5.7 Merge the different cores of a project (`merge-cores`)

Merges the entire synthesizable content of an project into a pair of VHDL/Verilog files

5.8 Create/update an FPGA project (`project`)

When a top manifest has been written for synthesis, `hdlmake` reads the targeted tool and creates a new specific project by adding both the whole file set from the module tree and the appropriated project properties.

The project will be specific for the targeted synthesis tool and, if this already exists, the `hdlmake` will update its contents with the ones derived from the module/files hierachy in the Manifest tree.

Currently, the following FPGA IDEs are supported:

Vendor	FPGA IDE
Xilinx	ISE
Xilinx	PlanAhead
Altera	Quartus II
Lattice Semi.	Diamond IDE
Microsemi (formerly Actel)	Libero IDE/SoC

Note: both `ise-project` and `quartus-project` commands has been mantained in the code for backwards compatiblity. In any case, when any of these are found, the general `project` action is launched.

5.9 Automatic execution (`auto`)

This is the default action for hdlmake, the one that is run when a command is not given.

The basic behaviour will be defined by the value of the `action` manifest parameter in the hierachy top `Manifest.py`. This can be set to `simulation` or `synthesis`, and the associated command sequence will be:

simulation:

1. generate a simulation makefile including all the files required for the defined testbench

synthesis:

1. create/update the FPGA project including all the files required for bitstream generation
2. generate a synthesis makefile

Note: in any case, it's supposed that all the required modules have been previously fetched. Otherwise, the process will fail.

CHAPTER 6

Manifest variables description

6.1 Top Manifest variables

action ; [<type ‘str’>] ; What is the action that should be taken (simulation/synthesis), default=”” top_module ; [<type ‘str’>] ; Top level entity for synthesis and simulation, default=None incl_makefiles ; [<type ‘list’>, <type ‘str’>]; List of .mk files appended to toplevel makefile, default=[]

6.2 Universal variables

fetchto ; [<type ‘str’>] ; Destination for fetched modules , default=None modules ; [<type ‘dict’>] ; List of local modules , default={} files ; [<type ‘str’>, <type ‘list’>]; List of files from the current module , default=[] library ; [<type ‘str’>] ; Destination library for module’s VHDL files , default=work include_dirs ; [<type ‘list’>, <type ‘str’>]; Include dirs for Verilog sources , default=None

6.3 Simulation variables

sim_tool ; [<type ‘str’>] ; Simulation tool to be used (e.g. isim, vsim, iverilog), default=None sim_pre_cmd ; [<type ‘str’>] ; Command to be executed before simulation , default=None sim_post_cmd ; [<type ‘str’>] ; Command to be executed after simulation , default=None

vsim_opt ; [<type ‘str’>] ; Additional options for vsim , default=”” vcom_opt ; [<type ‘str’>] ; Additional options for vcom , default=”” vlog_opt ; [<type ‘str’>] ; Additional options for vlog , default=”” vmap_opt ; [<type ‘str’>] ; Additional options for vmap , default=”“

iverilog_opt ; [<type ‘str’>] ; Additional options for iverilog , default=”“

quartus_preflow; [<type ‘str’>] ; Quartus pre-flow script file , default=None quartus_postmodule; [<type ‘str’>] ; Quartus post-module script file , default=None quartus_postflow; [<type ‘str’>] ; Quartus post-flow script file , default=None

sim_only_files ; [<type ‘list’>, <type ‘str’>]; List of files that are used only in simulation, default=[] bit_file_targets; [<type ‘list’>, <type ‘str’>]; List of files that are used only in simulation, default=[]

6.4 Synthesis variables

target ; [<type ‘str’>] ; What is the target architecture , default=”” syn_tool ; [<type ‘str’>] ; Tool to be used in the synthesis , default=None syn_device ; [<type ‘str’>] ; Target FPGA device , default=None syn_grade ; [<type 'str'>] ; Speed grade of target FPGA , default=None syn_package ; [<type 'str'>] ; Package variant of target FPGA , default=None syn_top ; [<type 'str'>] ; Top level module for synthesis , default=None syn_project ; [<type 'str'>] ; Project file name , default=None syn_ise_version; [<type 'type'>, <type 'str'>]; Force particular ISE version , default=None syn_pre_cmd ; [<type 'str'>] ; Command to be executed before synthesis , default=None syn_post_cmd ; [<type 'str'>] ; Command to be executed after synthesis , default=None

6.5 Miscellaneous variables

syn_name ; [<type 'str'>] ; Name of the folder at remote synthesis machine, default=None force_tool ; [<type 'str'>] ; Force certain version of a tool, e.g. 'ise < 13.2' or 'iverilog == 0.9.6, default=None

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 `--py ARBITRARY_CODE`

Add arbitrary code when evaluation all manifests

7.3 `--log LOG`

Set logging level for the Python logger facility. You can choose one of the levels in the following tables, in which the the associated internal logging numeric value is also included:

Log Level	Numeric Value
`critical`	50
`error`	40
`warning`	30
`info`	20
`debug`	10
not provided	0

7.4 `--generate-project-vhd`

Warning: this is an experimental feature!!

Generate `project.vhd` file with a meta package describing the project.

This option is targeted to VHDL designs in which the SDB (Self Describing Bus) standard is going to be used. You can get more information about SDB in the following link: http://www.ohwr.org/projects/fpga-config-space/wiki

7.5 `--force`

Force hdlmake to generate the makefile, even if the specified tool is missing.