RadioHDL

Speedup HDL development

User Manual

Authors:	Eric Kooistra Ruud Overeem
Version	0.2 dd. 11 december 2018

Content

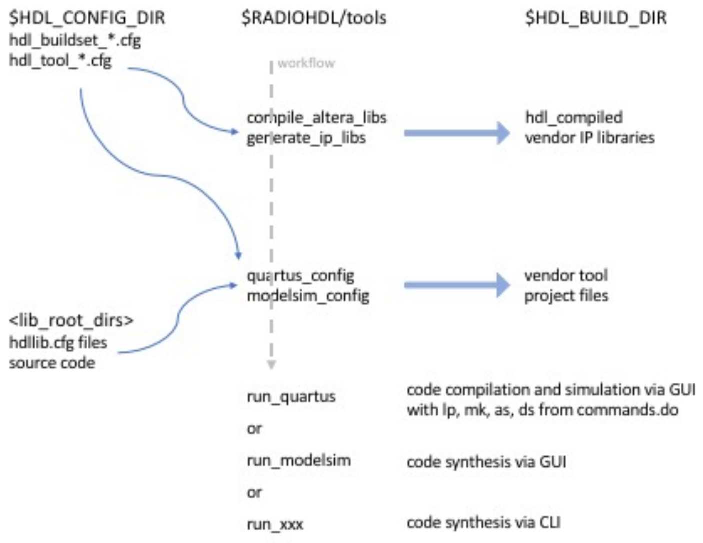
1 Introduction

The RadioHDL package offers tools for setting up and maintaining your development environment for programming FPGAs. Currently the package supports tools like modelsim, questasim and quartus but other tools can be added easily.

RadioHDL can be configured using three different kinds of configuration files:
- a hdl_tool file which describes where you have installed you vendor package
- one or more hdl_buildset files that describe the technologies of your board and which tools you want to use for developing your code.
- multiple hdllib files, each describing one library.

These configuration files are described in more detail later in this document.

After setting up your config files RadioHDL can create for your selected buildset all libs for simulation, create all IP libs, find out dependencies, generate project files and do a lot more of these labour-intensive tasks for you.



2 Setting up RadioHDL environment

After downloading RadioHDL from opencores.org you have to setup some configuration files before you can use RadioHDL for the first time.

In the main directory of your check-out there is a file called `init_radiohdl.sh`. This file defines some environment variables that are heavily used throughout the RadioHDL package. There are three environment variables that are important for using RadioHDL:
`RADIOHDL` : This variable will point to the main directory where your checked out the package. This is the location where this init_radiohdl.sh file is located. Do not change this variable.

`HDL_BUILD_DIR` : Points to the directory where RadioHDL will store all its result files. If this variable is not already set when init_radiohdl.sh is run then it will be set to point to `$RADIOHDL/build`.
`HDL_CONFIG_DIR` : Points to the directory where RadioHDL expects the configuration files. If this variable is not already set when init_radiohdl.sh is run than it will be set to point to `$RADIOHDL/config`.

Besides setting up some environment variables the init_radiohdl.sh script will extend your `PATH` and `PYTHONPATH` to include some paths within the RadioHDL package.

Examples

If you want to use the default values of the RadioHDL package it is enough to source the init_radiohdl.sh file:

```
cd <main_check_out_directory>
. ./init_radiohdl.sh
```

Is it likely however that you want to have your configuration files outside the checkout of the RadioHDL package in order not to mix your own stuff with the package.
In that case define the HDL_CONFIG_DIR variable before sourcing the init_radiohdl.sh file:

```
export HDL_CONFIG_DIR=<my_own_config_dir>
cd <main_check_out_directory>
. ./init_radiohdl.sh
```

Take care of all vendor specific environment variables in your own .bashrc or local .profile file. For example Altera and Modelsim need the key `LM_LICENSE_FILE` to be defined. So our .bashrc file contains the line:

```
export LM_LICENSE_FILE=<our_license@our_company>
```

2.1 Configuration file basics

Before diving into the two configuration files you have to set up to get RadioHDL working on your machine, first some basic information about how RadioHDL handles configuration files.

A configuration file is a collection of key–value pairs. Key and value are separated with the assignment char '='. Everything after (and including) a '#' char till the end of the line is treated as comment and is skipped in the interpretation of the files.

```
this_is_a_key = and this is the value    # except for this comment.
```

To make the files easier to read and maintain for humans each value can be defined in more than one line where spacing is trivial. This makes is easy to define values that consist of more than one item. Actually, the concept of 'value' is implemented as 'being everything between the assignment char and the start of the next key or the start of a section.
Newline characters are translated into a single space.

```
invitationlist = John & Marry,
                 Robert,
                 Sandra and Naomi,
```

This is equivalent with:

```
invitationlist = John & Marry, Robert, Sandra and Naomi,
```

Keys are always in lowercase even when the name of the key will be used as environment variable. RadioHDL takes care that the key names are converted to uppercase in these cases.

To make the files more readable the files can be divided into sections. A section marker is a line that starts with '[', contains a string and ends with ']'. Currently sections have no other function as to 'group' a collection of key-value pairs.

```
[ start of a new section ]
key = value
```

You can refer to a value of another key in the same configuration file by placing the name of that key between '<' and '>'.

```
tool_name     = quartus
tool_version  = 12.3
tool_location = /home/software/<tool_name>/<tool_version>/bin
```

2.2 Setting up a buildset configuration file

The first configfile you have to make is one that describes which (version of) tools your like to use and on which technologies your FPGA is based. This combination of technologies and tool(version)s is called a ***buildset*** in RadioHDL. The name of buildset files is `hdl_buildset_<buildset>.cfg` and they are, of course, located in `$HDL_CONFIG_DIR`.

Below is an example of a buildset file:

```
# Uniboard 1 configuration
buildset_name               = unb1
technology_names            = ip_stratixiv
family_names                = stratixiv
block_design_names          = sopc

# tool names and versions
synth_tool_name             = quartus
synth_tool_version          = 11.1
sim_tool_name               = modelsim
sim_tool_version            = 6.6c

# environment variables
lib_root_dirs               = $RADIOHDL/libraries
                              $RADIOHDL/applications
                              $RADIOHDL/boards

[quartus]
quartus_dir                 = /home/software/Altera/<synth_tool_version>

[modelsim]
modelsim_dir                = /home/software/Mentor/<sim_tool_version>/modeltech
modelsim_platform           = linux_x86_64
model_tech_altera_lib       = /home/software/modelsim_altera_libs/<synth_tool_version>

# modelsim libraries
modelsim_search_libraries =
    # stratixiv only
    altera_ver lpm_ver sgate_ver altera_mf_ver altera_lnsim_ver stratixiv_ver
    stratixiv_hssi_ver stratixiv_pcie_hip_ver
    altera     lpm     sgate     altera_mf     altera_lnsim     stratixiv
    stratixiv_hssi     stratixiv_pcie_hip
```

The first ten keys, `buildset_name` till `build_dir` are required keys. The other keys depend on the values you fill in for the keys `synth_tool_name` and `sim_tool_name`.

`buildset_name`
Free to choose name that identifies this buildset. The name is used in the directory structure that is created during the several build/compile stages. It turns out the be handy to make this value equal to the `<buildset>` value you use in the name of your buildset file.

`technology_names`
Lists the IP technologies that are applicable for your FPGA, eg. ip_stratixiv, ip_arria10, ip_arria10_e1sg. It is possible to define multiple technologies as long as your tool (eg. Modelsim) supports these technologies. RadioHDL will only build libs that are based on the technologies you define in this key.

`family_names`
Lists the family(s) your FPGA belongs to. For simulation only the libs for these families are compiled.

`block_design_names`
???

`sim_tool_name, sim_tool_version`

Name and version number of the simulation tool to use in this buildset. Note: RadioHDL will look for a key in this config file that has the name of the value of `sim_tool_name` extended with `_dir`. So in this example file RadioHDL expects a key with the name `quartus_dir`.

`synth_tool_name, synth_tool_version`
Name and version number of the synthesis tool to use in this buildset. Note: RadioHDL will look for a key in this config file that has the name of the value of `synth_tool_name` extended with `_dir`. So in this example file RadioHDL expects a key with the name `modelsim_dir`.

`lib_root_dirs`
Lists the root directories where RadioHDL will search for libraries. Note that you can use existing environment variables.

`quartus_dir`
The location where your quartus package is installed. Note that you can refer to others keys defined in this configuration file by placing the keyname between '<' and '>'.

`modelsim_dir`
Location of the modelsim.ini file.

`modelsim_platform`
Targeted platform for modelsim. Executables like vsim, vlog and other are located here.

`model_tech_altera_lib`
The location where your simulation libraries from Altera will be stored.

`modelsim_search_libraries`
List of IP technology search libraries that will be put in the -L {} option of a Modelsim simulation configuration in the mpf. This avoids that all IP technology needs to be compiled into the work library. The -L {} option is needed for simulations in libraries that use generated IP like ip_stratixiv_phy_xaui which do not recognize the IP technology libraries mapping in [libraries] section in the mpf. The -L {} option is added to all simulation configurations in all mpf even if they do not need it, which is fine because it does not harm and avoids the need for having to decide whether to include it or not per individual library or even per individual simulation configuration.

2.3 Setting up a tool configuration file

After setting up a buildset configuration file that suits your situation the next thing to do is to create a configuration file that contains tool specific values like paths and environment variables. You typically have one tool configuration file that covers multiple versions of this tool. The name of these configuration files is `hdl_tool_<toolname>.cfg`.

Below is an example for quartus.
```
# configuration file for defining the quartus installation on this system
# define required environment variables for quartus
quartus_rootdir          = ${QUARTUS_DIR}/quartus
quartus_rootdir_override = ${QUARTUS_DIR}/quartus
niosdir                  = ${QUARTUS_DIR}/nios2eds

# extension to the PATH variable
quartus_paths =
    <quartus_rootdir>/bin
    <niosdir>/bin
    <niosdir>/bin/gnu/H-i686-pc-linux-gnu/bin
    <niosdir>/bin/gnu/H-x86_64-pc-linux-gnu/bin
    <niosdir>/sdk2/bin

[sopc]
sopc_paths =
    <quartus_rootdir>/sopc_builder/bin
sopc_environment_variables =
    sopc_kit_nios2      <niosdir>
```

```
[qsys]
qsys_paths =
    <quartus_rootdir>/../qsys/bin

[ip generation]
ip_tools                         = qmegawiz qsys-generate quartus_sh
qmegawiz_default_options         = -silent
qsys-generate_default_options = --synthesis=VHDL --simulation=VHDL --allow-mixed-
language-simulation
quartus_sh_default_options       =

[user settings]
user_paths                    =
user_environment_variables =
    altera_hw_tcl_keep_temp_files       1
```

Although many keys in a hdl_tool configuration file are tool specific there are some keys RadioHDL will always search for. RadioHdl will always search for keys ending in `_paths` and `_environment_variables` for keys that begin with:

- the toolname of that file (eg. `quartus_paths` and `quartus_environment_variables`)
- the `block_design_names` you mentioned in your buildset file (like sopc, qsys)
- "`user`". So you can always define add your own paths and environment variables with the keys `user_paths` and `user_environment_variables`.

quartus_rootdir, quartus_rootdir_override and niosdir
Quartus needs these three environment variables in order to operate properly. Note that you can refer to environment variables that you have defined in your buildset file.

quartus_paths
List of paths that are needed to run quartus programs. Sometimes paths differ slightly between several versions of a tool. Just mention all paths here because paths that do not exist (for your <buildset> version) are not added to your PATH variable.

[<toolname>]
For the tools you mentioned in the key 'block_design_names' of your buildset file you can define two keys that are automatically recognized by RadioHDL. These keys are:
<toolname>_paths: the paths to add to your PATH variable.
<toolname>_environment_viariables: the environment variables this tool need. Note that key(=name of environment variable) and value(=value of environment variable) are separated with white-space.

[ip generation]
This section defines the tools and their options that will be used for the generation of the IP libraries. Currently the generate_ip_libs executable from RadioHDL supports three generation tools: qmegawiz, qsys-generate and quartus_sh.
For each of these tools you can define a key <toolname>_default_options that contains the default options to add to the commandline when running this tool. Options that are lib-specific are defined in the hdllib.cfg file. In those files the key <toolname>_extra_options can be used to add other flags/arguments to the IP compilation.

[user settings]
Finally you can define paths and environment variables that are specific for you own need. The keys user_paths and user_environment_variables are recognized by RadioHDL.

2.4 Check your configuration

Since there are some dependencies between the buildset- and the tool- configuration file there is a utility that checks (most) values in your files:

```
check_config buildsetname
```

Eg. 'check_config unb1' will check the content of the file hdl_buildset_unb1.cfg and the hdl_tool_xxx.cfg files it refers to.

Once your config files are correct you can start using (some of) the RadioHDL utilities.

2.5 Setting up library configurationfiles

To enable RadioHDL to do the generation of many files for you, you have to describe each library in its own `hdllib.cfg` file. RadioHDL uses these files to:

1. compile the library binaries for simulation
2. synthesize an image that can be loaded ion the FPGA
3. verify VHDL test benches in simulation

Although the creation of these hdllib.cfg files may take some time you will definitely save time on the long run.

The HDL library files can define:

- a module library with VHDL that is reused in other libraries
- a design library with a top level entity that maps on the IO of the FPGA.

For the hdllib.cfg there is no difference between a module library or a design library, they are all HDL libraries. The hdllib.cfg typically points to sources that are located in the same directory or in its subdirectories. However the sources can be located elsewhere, the hdllib.cfg can refer to sources at any location.

All your VHDL files are grouped into libraries. The rule is that each VHDL file is compiled in only one library. If a component is used in another library then it is instantiated using <hdl_library_clause_name>.<entity name>.

Note: The definitions made in the hdl_buildset_<buildset>.cfg can be used as keywords in the hdllib.cfg. To use these keywords, the key should be put between pointy brackets <>.
As an example, the hdl_buildset_unb1.cfg has the key '`buildset_name`' with a value: `unb1`.
If the string "`<buildset_name>`" occurs in the hdllib.cfg file, it will be replaced by "`unb1`" when running the RadioHDL utilities.

Sections

The hdllib.cfg files are used by many utilities from the RadioHDL package. To structure the keys in the hdllib.cfg file somewhat the file uses sections:

`[modelsim_project_file]`	: key-value pairs for modelsim_config
`[quartus_project_file]`	: key-value pairs for quartus_config
`[generate_ip_libs]`	: key_value pairs to generate the IP libraries

Future target scripts can have their own [section name] header in the hdllib.cfg to keep the files more organised.

Below is an example of a hdllib.cfg file.

```
hdl_lib_name            = ip_stratixiv_phy_xaui
hdl_library_clause_name = ip_stratixiv_phy_xaui_lib
hdl_lib_uses_synth      = common
hdl_lib_uses_sim        =
hdl_lib_uses_ip         =
hdl_lib_technology      = ip_stratixiv
hdl_lib_include_ip      =

synth_files =
    ip_stratixiv_phy_xaui_0.vhd
    ip_stratixiv_phy_xaui_1.vhd
    ip_stratixiv_phy_xaui_2.vhd
    ip_stratixiv_phy_xaui_soft.vhd

test_bench_files =
    tb_ip_stratixiv_phy_xaui.vhd
    tb_ip_stratixiv_phy_xaui_ppm.vhd


[modelsim_project_file]
modelsim_copy_files =
    wave_tb_ip_stratixiv_phy_xaui.do     .
    wave_tb_ip_stratixiv_phy_xaui_ppm.do .

modelsim_compile_ip_files =
    $RADIOHDL/libraries/technology/ip_stratixiv/phy_xaui/compile_ip.tcl
    $RADIOHDL/libraries/technology/ip_stratixiv/phy_xaui/compile_ip_soft.tcl


[quartus_project_file]
quartus_copy_files =
quartus_vhdl_files =
quartus_sdc_files  =
quartus_qip_files  =
    generated/ip_stratixiv_phy_xaui_0.qip
    ip_stratixiv_phy_xaui_soft.qip

[generate_ip_libs]
qmegawiz_ip_files =
    ip_stratixiv_phy_xaui_0.vhd
    ip_stratixiv_phy_xaui_1.vhd
    ip_stratixiv_phy_xaui_2.vhd
    ip_stratixiv_phy_xaui_soft.vhd
```

hdl_lib_name
The name of the HDL library, e.g. `common, dp, unb1_minimal`.

hdl_library_clause_name
The name of the HDL library as it is used in the VHDL LIBRARY clause, e.g. `common_lib, dp_lib, unb1_minimal_lib`.

hdl_lib_disclose_library_clause_names
If a component from a library is instantiated as a component (instead of as an entity) then that means that this library may be unavailable and in that case it has to be listed as a pair of lib_name and library_clause_name at this '`hdl_lib_disclose_library_clause_names`' key. For components that are instantiated as components the actual source library may have been removed (via the '`hdl_lib_technology`' key) or it may even not be present at all. The library clause name of instantiated components is used in the VHDL code at the LIBRARY statement in e.g. a tech_*.vhd file to ensure default component binding in simulation. The '`hdl_lib_disclose_library_clause_names`' key is then used in the hdllib.cfg file of that (technology) wrapper library to disclose the library clause name of the component library that is listed at the `hdl_lib_uses_*` key.

hdl_lib_uses_synth
List of HDL library names that are used in this HDL library for the 'synth_files', only the libraries that appear in VHDL LIBRARY clauses need to be mentioned, all lower level libraries are found automatically. The following libraries have to be declared at the '`hdl_lib_uses_synth`' key:
- Libraries with packages that are used
- Library components that are instantiated as entities

Libraries that are instantiated as components can be specified at the '`hdl_lib_uses_synth`' key, but instead it may also be specified at the '`hdl_lib_uses_ip`' key. If there are different source variants of the component and if these source libraries can be missing in the 'lib_root_dir' tree, then the library must be specified at the '`hdl_lib_uses_ip`' key.

hdl_lib_uses_sim
List of HDL library names that are used in this HDL library for the 'test_bench_files', only the libraries that appear in VHDL LIBRARY clauses need to be mentioned, all lower level libraries are found automatically.

The 'hdl_lib_uses_synth' and 'hdl_lib_uses_ip' keys and 'hdl_lib_uses_sim' key separate the dependencies due to the synth_files from the extra dependencies that come from the test bench files. Quartus can exit with error if IP is included in the 'hdl_lib_uses_ip' list of libraries but not actually used in the design, eg due to a sdc file that is then sourced but that cannot find some IP signals. Having a seperate 'hdl_lib_uses_ip' and 'hdl_lib_uses_sim' key solves this issue, by avoiding that libraries that are only needed for test bench simulation get included in the list for synthesis. Often the 'test_bench_files' do not depend on other libraries then those that are already mentioned at the 'hdl_lib_uses_synth' key, so then the 'hdl_lib_uses_sim' remains empty.

hdl_lib_uses_ip
The 'hdl_lib_uses_ip' typically defines IP libraries that have multiple variants even within a specific technology (as specified by buildset key 'technology_names'). However typically only one tech variant of the IP is used in a design. The 'hdl_lib_include_ip' key therefore defines the library that must be included in the list of library dependencies that are derived from 'hdl_lib_uses_ip'. Hence the 'hdl_lib_uses_ip' key defines the multiple choice IP libraries that are available in this library and the 'hdl_lib_include_ip' select which one (or more) are used by a higher level component (design). For tech libraries with only one IP library variant the IP libraries should be listed at the 'hdl_lib_uses_synth' key or at both the 'hdl_lib_uses_ip' and 'hdl_lib_include_ip' key. If a multiple choice IP library can be included always, then it may also be specified at the 'hdl_lib_uses_synth'.
Typically present, but unused IP is no problem. However for synthesis the constraint files of unused IP can cause problems.
Therefore then use 'hdl_lib_include_ip' to only include this IP library from the IP variants in 'hdl_lib_uses_ip'. An example is to only include ip_stratixiv_ddr3_uphy_4g_800_master in unb1_ddr3 / io_ddr / tech_ddr by setting hdl_lib_include_ip = ip_stratixiv_ddr3_uphy_4g_800_master in the hdllib.cfg of unb1_ddr3. Another example is ip_stratixiv_tse_sgmii_lvds for tech_tse which is included by the board specific library unb1_board to avoid that the other ip_stratixiv_tse_sgmii_gx variant is also include when it is not actually used. This example also shows that a 'hdl_lib_include_ip' can also occur at some intermediate hierarchical component level in a design. The advantage is that the include of ip_stratixiv_tse_sgmii_lvds in the unb1_board hdlib.cfg now automatically applies to all designs that instantiate unb1_board.
The exclusion can only be done when the component is instantiated as a component and not as a entity. Therefore the exclusion is done at the IP level, because the IP is instantiated as component. Hence the exclusion works because for a component instance that is not used, only the component declaration (in the component package) needs to be known by the tools. Hence the exclusion makes use of the same VHDL component mechanism as the technology independence.
The exclusion is only done for synthesis, so not for simulation. The reason is that for simulation it is oke to keep the library included.
The difference between this 'hdl_lib_uses_ip' key and the 'hdl_lib_technology' key is that the HDL libraries with 'hdl_lib_technology' key value that does not match the specified technologies are not build. Whereas HDL libraries that are excluded via the combination of 'hdl_lib_include_ip' and 'hdl_lib_uses_ip' are still created in the build directory, but they are not used for that HDL library so they are excluded dynamically.

hdl_lib_include_ip
The 'hdl_lib_uses_*' keys identify which libraries are available for that particular HDL library. For simulation they are all included. The 'hdl_lib_include_ip' identifies which IP libraries from 'hdl_lib_uses_ip' will actually be included for synthesis.
The 'hdl_lib_include_ip' typically appears in another higher layer HDL library. IP libraries can be included in the following ways:
- by listing the IP library name at the 'hdl_lib_uses_synth' key, then it is always included
- by listing the IP library name at the 'hdl_lib_uses_ip' key, and including it explicitly with the 'hdl_lib_include_ip' key.

The 'hdl_lib_include_ip' is typically set at:
- the design library that actually uses that IP library, this then has to be done per design revision.
- for IP in unb*_board that is used in all designs it is set in these unb*_board libraries so that it is then automatically included for all designs that use the unb*_board library (i.e. via ctrl_unb*_board.vhd).

Note that specifying an IP library at the '`hdl_lib_uses_ip`' key and then including it via '`hdl_lib_include_ip`' in the same hdllib.cfg, is equivalent to specifying the IP library at the '`hdl_lib_uses_synth`' key.

hdl_lib_technology
The IP technology that this library is using or targets, e.g. ip_stratixiv for UniBoard1, ip_arria10 for UniBoard2. For generic HDL libraries use ''. For simulating systems with multiple FPGA technologies it is also possible to list multiple IP technology names.

test_bench_files
All HDL files that are needed only for simulation. These are typically test bench files, but also HDL models. For Modelsim they need to be in compile order and they are placed in the 'test_bench_files' project folder.
Both Verilog and VHDL files are supported.

regression_test_vhdl
List of pure VHDL testbenches that need to be included in the regression simulation test. For Modelsim this key is used by modelsim_regression_test_vhdl.py to simulate all testbenches and report their result in a log. The VDHL test benches must be self-checking and self-stopping.

synth_files
All HDL files that are needed for synthesis. For Modelsim they need to be in compile order and they are placed in the 'synth_files' project folder.
For Quartus synthesis these files get included in the HDL library qip file.
Both Verilog and VHDL files are supported.

synth_top_level_entity
When this key exists then a Quartus project file (QPF) and Quartus settings file (QSF) will be created for this HDL library. If this key does not exist then no QPF and QSF are created. The '`synth_top_level_entity`' key specifies the top level entity in the HDL library that will be the top level for synthesis. If the key value is '' then the '`hdl_lib_name`' is taken as top level entity name.

* Created QPF:
- It only states that there is one revision that has the name of the 'synth_top_level_entity'. The Quartus scheme for revisions is not used. Instead the RadioHDL scheme of defining design revisions as separate HDL libraries is used.

* Created QSF:
- Defines the top level entity name using 'synth_top_level_entity'
- It sources the files listed by the 'quartus_qsf_files' key, this is typically a board qsf that defines settings that are common to all designs that target that board, eg. unb1_board.qsf.
- It sources all library QIP files <lib_name>_lib.qip that are needed by the design. The library QIP files are sourced in dependency order so that the top level design <lib_name>_lib.qip is sourced last. In this way the top level design constraints are at the end.

* Created <lib_name>_lib.qip files
The <lib_name>_lib.qip files are created for each library using the following keys in this order:
- hdl_lib_uses_synth -- used for all HDL libraries
- quartus_vhdl_files -- used for IP libraries that have different HDL file for sim and for synth (typically not needed for most IP)
- quartus_qip_files -- used for IP libraries (constaints for the IP), top level design libraries (SOPC or QSYS MMM, e.g. sopc_unb1_minimal.qip)
- quartus_tcl_files -- used for top level design libraries (pinning definitions, e.g. unb1_minimal_pins.tcl)
- quartus_sdc_files -- used for top level design libraries (timing constraints, e.g. unb1_board.sdc)

<tool_name>_copy_files
The copy_files key can copy one file or a directory. The first value denotes the source file or directory and the second value denotes the destination directory. The paths may use use environment variables. The file path or directory can be an absolute path or a relative path. The relative path can be

from hdllib.cfg location in SVN or from the build dir location. Whether the source directory is the hdllib.cfg location in SVN or the build_dir location depends on the <tool_name>. For modelsim_copy_files and quartus_copy_files the relative source directory is the hdllib.cfg location in SVN and the relative destination directory is the build_dir location. The direction can be from build dir to dir in SVN or vice versa, or to any directory location in case absolute paths are used. The destination directory will be removed if it already exists, but only if it is within in the build_dir.
If the destination directory is not in the build_dir then it first needs to be removed manually to avoid accidentally removing a directory tree that should remain (eg. ~).

modelsim_copy_files
Copy listed all directories and files for simulation with Modelsim, used when `sim_tool_name = modelsim` in hdl_buildset_<buildset>.cfg. Can be used to eg. copy wave.do or data files from SVN directory to the build directory where the Modelsim project file is. For data files that are read in VHDL the path then becomes data/<file_name>.

modelsim_compile_ip_files
This key lists one or more TCL scripts that are executed by the Modelsim mpf before it compiles the rest of the source code. E.g.:

- compile_ip.tcl : a TCL script that contains external IP sources that are fixed and need to be compiled before the synth_files. For the Altera IP the compile_ip.tcl is derived from the msim_setup.tcl that is generated by the MegaWizard or Qsys.
- map_ip.tcl : a TCL script that maps a VHDL library name to another location.

quartus_copy_files
Copy listed all directories and files for synthesis with Quartus, used when `synth_tool_name = quartus` in hdl_buildset_<buildset>.cfg. Can be used to eg. copy sopc or qsys file from SVN directory to the build directory where the Quartus project file is and that is where the run_* bash commands expect them to be.

quartus_qsf_files
See also '`synth_top_level_entity`' description.
One or more .qsf files that need to be included in the HDL library qsf file for Quartus synthesis of a '`synth_top_level_entity`' VHDL file.

quartus_vhdl_files
See also '`synth_top_level_entity`' description.
One or more .vhdl files that need to be included in the HDL library qip file for Quartus synthesis. These are VHDL files that must not be simulated so they are not listed at the '`synth_files`' key. This can typically occur for technology IP libraries where e.g. a .vhd file is used for synthesis and a .vho file for simulation like in the tse_sqmii_lvds HDL library.

quartus_qip_files
See also '`synth_top_level_entity`' description.
One or more .qip files that need to be included in the HDL library qip file for Quartus synthesis.

quartus_tcl_files
See also '`synth_top_level_entity`' description.
One or more .tcl files that need to be included in the HDL library qip file for Quartus synthesis.

quartus_sdc_files
See also '`synth_top_level_entity`' description.
One or more .sdc files that need to be included in the HDL library qip file for Quartus synthesis.

2.6 user_components.ipx

It is important that the quartus tools look in the right directories when searching for library files. Altera invented the `user_components.ipx` file where you can specify your paths.

For RadioHDL we need the path: `$RADIOHDL/libraries/**/*` to be in this file. In the `tools/quartus` directory of RadioHDL is a file named `minimal_user_components.ipx`. Almost every utility of RadioHDL initializes the quartus environment and during this initialisation it checks if the user_components.ipx file on your system contains these minimal paths. If not than the utility stops with an error that tells you what file to modify.

NOTE: *Altera says that you can use your personal user_components.ipx file in ~/.altera.quartus/ip/<quartus_version>/ip_search_path for this kind of additional paths but tests show that this does not work for all tools of Altera. If the RadioHDL tools don't start please follow the message the tool gives you to fix the problem.*

3 Using the RadioHDL environment

3.1 Compiling the necessary libraries

3.1.1 Compiling the Altera libraries for simulation with Modelsim

The Altera verilog and vhdl libraries for the required FPGA device families can be compiled using:

```
compile_altera_simlibs <buildset>
```

For Modelsim versions newer than about version 10 this compile_altera_simlibs script must be used and not the tools/Launch simulation library compiler in the Quartus GUI, because the libraries have to be compiled with the 'vlib -type directory' option to be able to use 'mk all' in ModelSim.

For example to make all modelsim libraries for your 'unb1' buildset, type:

```
compile_altera_simlibs unb1
```

The script builds the libraries in $RADIOHDL/build directory, but modelsim expects them to be in the install directory of modelsim itself. So the script ends with an instruction you should execute to move the build libraries to the right place with sudo rights.
This instruction is something like:

```
sudo mv $RADIOHDL/build/quartus/11.1 /home/software/modelsim_altera_libs
```

3.1.2 Generate IP libraries

The IP needs to be generated before it can be simulated or synthesized. The Quartus IP is generated using Qsys or the Megawizard.

```
generate_ip_libs <buildset>
```

For example to make all IP libraries for your 'unb1' buildset, type:

```
generate_ip_libs unb1
```

NOTE: *The Altera tools make use of the `/tmp` directory intensively. Unfortunately these tools do not cleanup the files they created there. We ran into major problems when working with more than one user on the same machine:*
1) `/tmp/compute_pll_temp` and `/tmp` must have write-access for anyone (rwxrwxrwx)
2) run '`rm -rf /tmp/alt* /tmp/public*`' before running `generate_ip_libs`

Currently `generate_ip_libs` supports three IP generation tools: `qmegawiz, qsys-generate quartus_sh.`
These IP tools are configured in the `hdl_tool_quartus.cfg` file (key `ip_tools`) and implemented as special functions in `generate_ip_libs`. If you need different IP generation tools you should add them to the `ip_tools` key and add an extra function in `generate_ip_libs`.

3.2 Configuring Quartus

3.2.1 Creating the Quartus project files

The utility `quartus_config` creates the Quartus qpf, qsf and or qip files for a design library:

```
quartus_config <buildset>
```

For example to create all Quartus projectfiles that are related to your 'unb1' buildset, type:

```
quartus_config unb1
```

When a new VHDL file is added to a library or if a filename is changed, then it is necessary to rerun `quartus_config`. Typically it is not necessary to do delete the entire `HDL_BUILD_DIR` library of the design directory, but it can be necessary to make sure that Quartus is not 'confused' by obsolete files in that directory. Make sure that any image files that need to be kept are saved elsewhere, before deleting the build directory of the design library.

3.2.2 Top level design library

A `synth_top_level_entity` key in a library indicates that the library is a design library that can be synthesized to yield an FPGA image. A design library should not depend on another design library, because then there may occur conflicting or double design constraints for Quartus. Therefore it is not possible to reuse VHDL from a design library in other libraries. A solution is to put these VHDL files in a separate library. Another solution is to break the rule that a VHDL file should only be compiled once in one library and then also compile it in the design library that needs to reuse it.

3.3 Create control interface of your project

Dependent of which tool you like to use you can run 'run_qsys' or 'run_sopc' to generate the control interface of your design. Syntax:

```
run_qsys <buildset> <project> [<qsysfile>]
```
or
```
run_sopc <buildset> <project> [<sopcfile>]
```

We will limit the description of these two utilities to run_qsys since both utilities only differ in name not in functionality.

`<project>` is a directory that needs to exist under `$HDL_BUILD_DIR/${buildset}/quartus`. If no `<qsysfile>` is specified during the invocation the utility will look for a file with the name `<project>.qsys` in the project directory.

After generating all necessary file the utility will also build the nios application.

NOTE: *The sopc_builder from Altera uses Java and Java expects that it is started from an Xterm (environment `DISPLAY` must be set) otherwise it stops with an error.*

3.4 Running Modelsim

RadioHDL has three utilities that support you in running Modelsim: `modelsim_config`, `run_modelsim` and `modelsim_regression_test_vhdl`.

3.4.1 Creating the Modelsim project files

The binaries for Modelsim are build in a separate directory tree under `$HDL_BUILD_DIR`.
Use `modelsim_config` to create the Modelsim project files for all your HDL libraries:

```
modelsim_config <buildset>
```

The files this utility makes are stored in `$HDL_BUILD_DIR/<buildset>/modelsim`

Optionally you can clear this directory before running modelsim_config because everything is recreated:

```
 rm -rf $HDL_BUILD_DIR/<buildset>/modelsim
```

See also the docstring help text in the Python code:

```
> python
>> import modelsim_config
>> help(modelsim_config)
```

3.4.2 Compilation and makefiles

To support your work inside modelsim RadioHDL comes with a command.do file that contains a lot of usefull modelsim commands. To use these commando with your buildset start modelsim with:

```
run_modelsim <buildset> &
```

In Modelsim do:

```
lp eth               -- load eth library Modelsim project file
lp all               -- reports all libraries in order that eth depends on
mk compile all       -- compiles all libraries in order that eth depends on
mk compile           -- compiles only this eth library
```

To load another project do e.g.:

```
lp common
lp all               -- reports all libraries in order that common depends on
```

Instead of '`mk compile`' one can use Unix '`make`' and Modelsim '`vmake`' via the '`mk`' command. The advantage is that after an initial compile all any subsequent recompiles after editing a VHDL source file only will require recompilation of the VHDL source files that depend on it.

```
lp eth               -- load eth library Modelsim project file
mk clean all         -- deletes all created work directories and makefiles in $HDL_BUILD_DIR that
                        were needed for eth
mk all               -- makes all libraries in order that eth depends on and creates the makefiles
mk                   -- makes only this eth library
mk <lib_name>        -- makes only the specified <lib name> library
```

The first time '`mk`' is called the library is compiled and the library makefile is made. The library makefile is stored in the library build directory. It is important that the library compiles OK, because otherwise the library makefile is not created properly. Therefore keep on doing '`mk compile`' until the library compiles OK. Then when it compiles OK do '`mk clean`' to clear the library build directory at `$HDL_BUILD_DIR`, and then do '`mk`' to compile the library again and create a proper makefile. With the proper makefile it is sufficient to use '`mk`' to automatically recompile only the VHDL source files that changed or that depend on the changed file. Similar with '`mk all`' a change in some lower level library VHDL file will only cause that the VHDL files that depend on it will be recompiled. For a big project with many libraries using '`mk all`' is much easier and faster to use than '`mk compile all`'. Doing '`mk all`' and again '`mk all`' should show that the second '`mk all`' did not need to recompile any VHDL again.

Note that the eth library also depends on the dp library. Therefore when doing the following:

```
 lp dp
 mk all
 lp eth
 mk all
```

Then the '`mk all`' in the eth library will not recompile the libaries that were already compiled by the '`mk all`' in the dp library. Similar if you then do:

```
 mk clean dp
 mk all
```

Then the '`mk all`' in the eth library will recompile and recreate the makefile for the dp library, and then recompile the VHDL files in the higher libraries (up to eth) that directly or indirectly depend on a VHDL file in the dp library.
If a new VHDL file is added to a library or if a filename is changed, then it is necessary to rerun `modelsim_config` for simulation and to do '`mk clean`' on that library. It is important that Modelsim does not have that library open already, so either quit Modelsim first or '`lp`' to another library first, before running `modelsim_config`.

Typically it is not necessary to do '`mk clean all`', nor is it necessary to delete the entire `$HDL_BUILD_DIR` library subdirectories.

3.4.3 Simulation

The simulation is done using a VHDL test bench. These test bench files can be recognized by the 'tb_' prefix andhave a simulation configuration icon in the Modelsim GUI. To simulate a tb do e.g.:

double click tb icon, e.g.: `tb_eth`

```
as 10        -- add all signals of 10 levels deep into of tb hierarchy to the Wave Window
run -a       -- run all until the tb is done
```

The tb in the eth library runs as long as needed to apply the stimuli and they are self checking. The tb are instantiated into multi test bench tb_tb_tb_eth_regression.vhd. By running this tb_tb_tb_eth_regression the entire eth library gets verified in one simulation.

To change project library using '`lp <lib name>`' first the current active simulation needs to be closed using '`quit -sim`'.

To quit Modelsim without having to acknowledge the pop up do '`quit -f`'.

3.4.4 Regressiontests

TODO

3.5 Running Quartus

The best way to start quartus is using the script run_quartus. This script sets up the environment variables that match your buildset and then start quartus itself:

```
run_quartus <buildset>
run_qcomp.  ???
```

run_rbf ????

run_xxxx ????

4 Miscellaneous

4.1 Directory structure example

4.1.1 RadioHDL package

The RadioHDL package consists of the following directories:

applications: Contains firmware application designs, categorized by project.

```
applications/<project_name>/designs/<design_name>/revisions/<design_name_rev_name>
                                                 /quartus
                                                 /src
                                                 /tb
```

boards: Contains board-specific support files and reference/testing designs
Subdir 'designs' contains application designs that can be run on that board to test board-specific features.
Subdir 'libraries' contains board-specific support files, such as firmware modules to communicate with board-specific ICs, constraint files, pinning files, board settings template files.

```
boards/<board_name>/designs/<design_name>/quartus
                                         /src
                                         /tb
                   /libraries/<???>/quartus
                                         /src
                                         /tb
```

config: Contains the configuration files of RadioHDL that describe your buildsets and tools like quartus and modelsim. You can use another directory instead of this one by setting the environment variable `HDL_CONFIG_DIR` to another directory.

```
config
```

libraries: Several kinds of libraries are collected here:

- Library of reusable firmware blocks, categorized by function and in which generic functionality is separated from technology. Within technology another separation exists between generic technology and hardware-specific IP.
- The subdir 'external' contains HDL code that was obtained from external parties (e.g. open source).
- The subdirs 'designs' contains reference designs to synthesize the library block for specific boards.

```
libraries/base/      <library_name>/designs
         /dsp/       <library_name>/designs
         /external/  <library_name>/designs
         /io/        <library_name>/designs
         /technology/<library_name>/designs
```

software: Intended for software that runs on a PC, such as control/monitoring of boards and programs to capture and process board output, e.g. sent via Ethernet to the processing machine.

```
software
```

tools: Contains the utilities of RadioHDL.

```
tools/bin
     /modelsim
     /quartus
     /python
```

Several subdirectories can reoccur:

- src/vhdl : contains vhdl source code that can be synthesised
- tb/vhdl : contains vhdl source code that can is only for simulation (e.g. test benches, models, stubs)
- quartus : synthesis specific settings for design that uses Quartus and an Altera FPGA
- vivado : synthesis specific settings for design that uses Vivado and an Xilinx FPGA
- revisions : contains revisions of a design that only differ in generic setting

The separation of src/vhdl and tb/vhdl VHDL files is not mandatory, but can be convenient. An alternative would be to keep all VHDL in one vhdl/ sub directory. The hdl_lib_uses_synth key in hdllib.cfg typically contains the files from src/vhdl and the hdl_lib_uses_sim key typically contains the files from tb/vhdl.
The synthesis will only see the VHDL files that are listed at the hdl_lib_uses_synth key, because the files at the hdl_lib_uses_sim key are not needed for synthesis and could even confuse synthesis (e.g. warnings that file IO ignored because it is not possible to synthesize).

4.1.2 RadioHDL build directies

IP libraries make by generate_ip_libs(?)
```
build/quartus/<quartus_version>/vhdl_libs
                                verilog_libs

build/<buildset>/qmegawiz
                /quartus/<hdl_lib_name>
                /quartus_sh
```

TODO together with Eric

4.2 Design revisions

Within a design, several revisions can be made:
Add a directory 'revisions/' in the design directory which contains a list of subdirectories. Each subdirectory is
a revision. The design 'unb1_minimal' can be uses as an example. See the following revisions:

```
  designs/unb1_minimal/revisions/unb1_minimal_qsys/
  designs/unb1_minimal/revisions/unb1_minimal_sopc/
```

Each revision should at least have a 'hdllib.cfg' file and a toplevel .vhd file. See for example:

```
  unb1_minimal_qsys/hdllib.cfg
  unb1_minimal_qsys/unb1_minimal_qsys.vhd
```

In the toplevel vhdl file you can specify the 'g_design_name' generic (in this example 'unb1_minimal_qsys').
And in 'hdllib.cfg' you specify the libraries and keys you need, in this case 'unb1_minimal'.

When a design library has revisions, then the base library should not include keys for synthesis. Instead
the synthesis is only done for the revisions. The base library contains all the VHDL src and tb files. The revison hdllib.cfg should not refer to these base library VHDL files (i.e. using ../../src/vhdl), but instead they should instantiate the toplevel base entity using the base desing library name. This to follow the rule that a VHDL file is only compiled in one library.

5 Adding your own tools

The RadioHDL package that is distributed through opencores.org is based on the tools Modelsim, Questasim and Quartus. If you are using the same tools than everything should work in your environment after you checked/modified the configuration files in `$HDL_CONFIG_DIR` (see sections 2.2 till 2.4). But what if you are using others tool like eg. Vivado?
In that case you have to create some new configuration files, some directories and (unfortunately) extend one script. This chapter guides you through this process.

5.1 New hdl_tool_<toolname>.cfg file

Assume you want to add Vivado to RadioHDL: create in `$HDL_CONFIG_DIR` a file named `hdl_tool_vivado.cfg`. and put in that file Vivado specific paths and environment variables. There are no required keys in configuration files of the type hdl_tool so you are free to choose your own key names.
Note: the script you create in the following section uses the definitions you create in the hdl_tool file.

5.2 New <toolname> directory

Every tool has its own directory in `$RADIOHDL/tools` (see section 4.1.1). In this directory we expect a script named `set_<toolname>` to be present that accepts the buildset name as an argument. That script sets up the whole environment so that that tool can be used: extend paths, setup environment variables, etc.
Look in e.g. set_quartus how to get access to the information that is stored in your configuration files in the `$HDL_CONFIG_DIR`.

So to add e.g. to add Vivado:
- create a directory `$RADIOHDL/tools/Vivado`
- inside this directory make a script `set_vivado` that sets up the Vivado environment using the information that is configured in the hdl_buildset and hdl_tool files.

5.3 Extend generate_ip_libs

Currently there is one place (left) where tool specific code is part of a RadioHDL script and that is generate_ip_libs. In one of the future releases of RadioHDL this will be solved in a more generic way but for now you have to add a function to generate_ip_libs when you are using a tool other than `qmegawiz`, `qsys-generate` or `quartus_sh` for creating your IP libraries.

The main flow of generate_ip_libs is:
- read the hdl_buildset file of the buildset you started the program with.
- search for hdllib.cfg files in all directories mentioned in the key `lib_root_dirs`
- read the hdl_tool_<toolname>.cfg file where toolname is the value of the key `synth_tool_name` in your buildset file.
- read the value of the key `ip_tools` in this hdl_tool file which is a list of tools you use for creating the IP libraries. (each of these ip_tools have their own `<ip_tool>_default_options` key to configure the default options of that command.)
- finally it loops over all `buildset.technology_names`, all `tool.ip_tools` and all hdllibs that where read in to see if there is a match in technology and ip_tool with the information in the hdllibs.
- for each match found in these nested loops it calls a function called `run_<ip_tool_name>` that is part of the generate_ip_libs program.

So if you are using other ip_tools than the three mentioned above you have to add a `run_<your_ip_tool>` function to generate_ip_libs and extend the if-then-else construction in the main loop with a call to that function. And you have to setup your configuration files in the right way of course.

Take a look at the other run_ functions in generate_ip_libs what to implement. Basically it constructs a shell script that runs the IP tool and analyses the exit-code of the execution. No rocket science.

Index