RadioHDL firmware directory structure and development tools

	Organisatie / Organization	Datum / Date
Auteur(s) / Author(s): Eric Kooistra Harm Jan Pepping Daniel van der Schuur	ASTRON	
Controle / Checked: Andre Gunst	ASTRON	
Goedkeuring / Approval: Andre Gunst	ASTRON	
Autorisatie / Authorisation: **Handtekening / Signature**	ASTRON	

Distribution list:

Group:	Others:
Andre Gunst Harm-Jan Pepping Daniel van der Schuur Eric Kooistra	Gijs Schoonderbeek

Document history:

Revision	Date	Author	Modification / Change
0.1	2014-03-05	Eric Kooistra	Creation
0.2	2014-03-17	Eric Kooistra	Updated using ideas from Gaisler GRLIB and discussions with Harm Jan Pepping
0.3	2014-04-9	Eric Kooistra	• Extended contents to include not only the directory structure but also the development tools and scripts. • Defined a technology independent HDL development approach based on the two level approach from the Gaisler GRLIB. • Proposed /RadioHDL directory structure as new project in the current UniBoard_FP7 repository.
0.4	2014-04-10	Eric Kooistra	• Defined scheme for technology independent HDL mapping for UniBoard^2.
0.5	2014-04-18	Eric Kooistra	• Upated after review by HJP,DS and AG. • Rename ‘module’ into ‘library’ • Corrected description of using component instances for technology independent HDL • Added a preliminary hdl_lib_info.txt scheme for building all HDL library targets in /RadioHDL • Do not support SOPC in /RadioHDL, so start new designs and reproduce old designs with Qsys. • Assume /RadioHDL is a new repository that will also support the current designs for UniBoard1

ASTRON

Table of contents:

Terminology:

DP	Data Path streaming interface
FW	Firmware
HDL	Hardware Description Language
HVL	Hardware Verification Language
HW	Hardware
LCU	Local Control Unit (computer)
MM	Memory Mapped interface
ST	Streaming interface
SVN	Subversion Version Control
SW	Software
Tb	Test bench (VHDL)
Tc	Test case (Python)
UNB	UniBoard
Component	HDL component
Design	Top level HDL component that can run on an FPGA
Library	Library of HDL components
Module	HDL hierarchical component

References:

1. https://svn.astron.nl/UniBoard_FP7/ Repository for UniBoard1 applications
2. http://www.gaisler.com, “GRLIB IP Library User’s Manual”, Version 1.3.4 - B4140, December 2013
3. ASTRON-RP-1354, “UniBoard Firmware Compilation Guide: Reference design unb_minimal”
4. ASTRON-SP-056, “One-Click Design flow Specification”

1 Introduction

For UniBoard1 developments we have the UniBoard_FP7 repository. For the UniBoard^2 developments we need to restructure the way we work in the repository to be able to reuse the existing VHDL, Python and other source code (e.g. build scripts, …). Furthermore the repository should also fit the new development features of the OneClick environment like code generation, functional modelling and supporting multiple boards (e.g. Roach3, …). These new developments can be done within the current UniBoard_FP7 repository without causing problems to existing UniBoard1 designs. Therefore it appears not necessary to start a new repository for UniBoard^2 and the OneClick based developments.

2 Firmware repository directory structure

2.1 Why a repository?

For the firmware development we use a SVN repository as database to:

- group the source files that are needed for the firmware development
- have version control of source files
- support multi-user development by multiple designers
- support multi-site development by different institutes

2.2 What is in the repository and what is not?

To investigate what should be in the firmware repository and where in the repository it should be put, it is useful to think in terms of resources (or sources) and targets (or results, products). The repository contains the resources that are needed to create the targets of the firmware. Table 1 and Table 2 give a list of resources and Table 3 gives a list of targets. Not all resources are in the repository and these are listed in Table 2.

	Resources in the SVN repository
1	Board descriptions like pin files, interconnect HDL model for the boards, HDL model of an ADC, pdf of the schematic, … (Uniboard1, UniBoard^2, Roach3, ADU, PAC, AUB backplane, …)
2	Reference designs (test image, minimal image, io validation images, …)
3	Application designs (Aperitif BF + X, Arts BF, Aartfaac X, EVN X, Digital receiver, …)
4	Reusable VHDL libraries (base, dsp, io, …)
5	VHDL sources and other HDL like Verilog, Open CL, MyHDL, …
6	Board support package (board control interface via 1GbE or 10GbE, board control master, MM bus, SOPC, QSYS, board common peripherals like PPS, sensors, test IO, …)
7	Software for an embedded uP (unb_osy on NiosII board control master, Borph on Power PC, …)
8	Project files for simulation (modelsim: *.mpf, *.do) and synthesis (quartus: *.qsf, *.qip)
9	Build scripts for simulation (Modelsim: commands.do, libraries.ini, unb_msim), synthesis (Quartus: unb_*, unb_libraries.qip, user_components.ipx, embedded software: unb_app), HDL code generation (unb_mgw, …), hex file generator, …
10	One-click Python tool (register generator, code structure generation, modelling, …)
11	LCU memory mapped monitoring and control driver (node_io, uniserver, …)
12	LCU application driver (beamserver, UDP capture, …)
13	Algorithms, data analysis, plotting (in Matlab or Python, reorder map calculation, SNR, …)
14	Documentation (specification, readme, …)

Table 1 Resources in the SVN repository that are needed to achieve the targets

	Other resources outside the SVN repository
1	Python distribution (Anaconda, MyHDL,GNU radio, …)
2	Gate level technology HDL libraries (Altera, Xilinx)
3	Synthesis tool (Quartus, ISE)
4	Simulation tool (Modelsim)
5	Operating system (Linux, Windows)
6	PC hardware (LCU, dop233, …)
7	Board hardware
8	Engineers (private directory for each user)

Table 2 Other resources outside the SVN repository that are needed to achieve the targets

	Targets	Description
1	Application functional model simulation	Functional specification
2	Application FPGA images	E.g. a sof for an Altera FPGA
3	Application one or multi node validation result	Test case to prove the application
4	Design MM register map	E.g. in system.h file or systeminfo
5	HDL component reference design	Validated example design to prove a component on FPGA hardware (typically an IO component e.g. DDR3, 10GbE)
6	HDL component pass/fail simulation result	Test case to prove the component
7	Continuous integration (regression test) pass/fail result	List of test cases to prove the parts
8	Documentation	Publication (article, presentation, …)

Table 3: Targets that are the project deliverables or intermediate targets

2.3 How to organise the repository?

2.3.1 Trunk, branches and tags

For the UniBoard1 firmware developments it has appeared practicable to do all developments on the trunk. This scheme of developing directly on the trunk is also suitable for UniBoard^2.

For the UniBoard1 developments every application project directory has a local trunk, branches and tags directory. The Lofar repository for software development uses only one central trunk, branches and tags directory. The advantage of one central trunk is that a branch or a tag also contains directories and therefore also all dependencies that may exist between any directories.

2.3.1.1 Task branches

For the Lofar software development no work is done directly on the trunk. Each engineer works on a task branch in the branches directory. The changes are done on the task branch, verified with the regression tests and then merged to the trunk. After the merge the task branch is removed. This scheme of task branches ensures a stable trunk and allows multiple engineers to work on the same files. It does require the extra effort of managing the task branches and (manually) merging the changes to the trunk.

2.3.1.2 Release branches

In future a release branch for the RadioHDL/trunk could also contain a continuing development, e.g. for UniBoard1 developments when UniBoard1 applications only need to be maintained and become difficult to support with new versions of the reusable sources. A release branch typically exists forever. The Lofar repository also contains release branches.

2.3.1.3 Release tags

In SVN a tag is not really different from a branch. The difference in usage is that contrary to a branch on a tag no more commits should be done. Hence a tag provides a labelled name to a certain SVN revision number, because SVN has a global revision number that increments with every commit anywhere in the repository. A release tag typically exists forever.

2.3.2 Point of view

A grouping of files per sub-directory that is good from one point of view may result in files being scattered throughout the entire repository from another view. Files can be grouped in various ways e.g.:

1. Per type (e.g. keep Python files together)
2. Per HDL library based on their dependencies (e.g. keep HDL, python, reference design together in a sub directory)
3. In time (e.g. keep photo’s together per year)
4. …

For example if the Python code for a HDL peripheral is kept in the HDL library directory then the python peripheral files are no longer kept together. This then means that a check out of only the python PC host software is not enough to access a board, because all the firmware is then also needed to have the Python peripherals. For the UniBoard1 development the Python peripherals are kept together in a /peripherals directory.

Files that apply to various other groups ought to be stored at a higher level in the repository directory tree. This applies e.g. to build scripts, LCU monitoring and control driver, HDL libraries.

2.3.3 Directory hierarchy

Increasing directory hierarchy by adding sub directories enforces relational choices. For example if a system consist of two types of designs then these designs may be kept in sub directories of that system, e.g.:

```
/designs/
/systems/aperitif_beamformer/bn_filterbank
                            /fn_beamformer
        /aperitif_correlator/bn_channelizer
                            /fn_correlator
```

but they may also be kept at the same level as that system, e.g.:

```
/applications/<project_name>
             /apertif/designs/bn_filterbank
                             /bn_channelizer
                             /fn_beamformer
                             /fn_correlator
                     /systems/aperitif_beamformer
                             /aperitif_correlator
```

The disadvantage of using a more flat directory structure is that it may become unclear when it gets cluttered with many files. The disadvantage of adding hierarchy is that it suggests a fixed relation between the sources which may be too strict in future. For UniBoard1 development the /modules (HDL libraries), /designs and /systems directories are kept in parallel at the same level. In fact having these three directories still also allows defining HDL designs or even HDL libraries within a systems directory. If the designs and systems will be placed in an /applications/<project_name> directory then the flat scheme is preferred, because the

/<project_name> directory level already ensures that the /designs directory will not get cluttered with too many designs.

The more likely a resource will be reused the higher it needs to appear in the directory hierarchy. Adding support for a new board to the repository or adding a new application should affect only one high level directory. Expanding a resource with more variations of it should only have local affects in the repository.

2.3.4 Directory names

Only resources and targets appear in the directory structure.

1. Preferably use lower case, unless upper case is more clear.
2. No spaces for directory names, because that avoids the need to put the name in quotes.
3. It is not necessary to have institute names in the directory structure.
4. It is not necessary to have names of engineers in the directory structure.
5. Instead of using vendor names like Altera, Xilinx, it seems sufficient to directly use the device name (stratixiv, virtex4, …) or the tool name (quartus, ise, …) as directory name.
6. Typically language name should not be used as high level directory names, because is it really essential that the source is written in C or Python or Erlang and what if a mix of languages is used. For a sub directory a language name may be suitable e.g. /src/vhdl, but even then /src/hdl may be a better name to group HDL that will get synthesized, because it then may also contain Verilog code.
7. …

2.3.5 Reuse or copy

A change in reused sources affects all applications that use it. For HDL that provides the basic functionality, interfaces and IO components this is desirable, because this HDL is well tested and stable. The continuous integration (regression) tests will verify the modifications for all the applications that depend on it.

Separating the general functionality from the application specific functionality can be a lot of effort and not clear from the start. Initially a HDL or Python source could therefore also start as an application specific source. If later on it appears useful for other applications as well then it should be restructured for reuse and promoted to a more central library. This is for example how the general purpose ss_parallel.vhd has replaced the design specific bn_filterbank_ss.vhd.

When adding a feature to an existing reusable source it can be difficult to maintain backwards compatibility. If after reconsideration the new feature is still essential then this could result in a modified copy of the top level file (e.g. as with dp_offload_tx_dev.vhd) or an entire new HDL library directory. Dependent on the reusability of this copy it can exist in the same library directory or in a board or application specific directory. Within the repository one can make a SVN copy to preserve the history relation to the original source or actually copy the source and SVN add and commit it. Note that a branch is in fact an SVN copy of the trunk.

2.3.6 Overview of the GRLIB directory structure

Table 4 shows the top level directory structure of the Gaisler library (GRLIB) [2]. Typically a design that uses the GRLIB contains a Leon3 softcore processor and an AMBA bus to connect to the memory mapped peripherals. Together the program that runs on the Leon3 and the peripherals define the application. The GRLIB is set up such that it can map on many different FPGA or ASIC technologies.

```
/grlib-gpl-1.3.4-b4140               # = $GRLIB
      /lib/                          # HDL libraries for reuse
          /fmf
          /opencores
          /grlib
          /…
          /tech                      # FPGA technology IP simulation models
          /techmap/                  # Generic to IP wrappers per vendor
                  /inferred
                  /altera_mf         # RAM, PLL, mult
                  /stratix_iii       # DDR2
                  /…
                  /gencomp           # Generic IP components in gencomp.vhd
      /software                      # Embedded software for Leon3
      /bin                           # Global Makefile
      /boards/                       # FPGA boards
      /designs/                      # Applications for the boards
```

Table 4: Gaisler directory structure top level [2]

- /bin : has no sub directories and its main file is the global Makefile that finds all files via the list files.
- /boards : contains a sub directory for each supported board /<manufacturer_board_fpga>. The board directory typically contains a pin file (in the qsf) and constraints file (sdc) and a Makefile.
- /designs : contains the application designs for a board. This directory typically is flat and contains a Makefile, a top level entity file, a config.vhd, and one testbench.vhd, a boot program and a test program for the Leon3 processor.
- /software : Embedded software for the Leon3 processor with test programs for the peripherals. These tests can also run in HDL simulation. The GRLIB does not seem to provide PC host software e.g. to control the board.
- /lib : the gencomp.vhd contains the components of the generic IP wrappers and a list of constants numbers that identify the supported FPGA or ASIC technologies.

2.3.7 Using lists of files or find commands

Files that are scattered through the repository but that still also need to be used together can be gathered using:

1. list files
2. find commands
3. symbolic links (Linux)
4. SVN externals
5. …

The advantages of a predefined list file are that it contains only the relevant sources and that it can also defines the order in case the source files need to be processed in a certain order. The advantage of using find commands is that it is not necessary to update a list file when a new source is added. Table 5 shows the various lists of sources that are already being used for the UniBoard1 developments.

Files list	**Task**	**Description**
documents.html	Bundle	Central access point to various documents
libraries.ini	Compile (Modelsim)	HDL libraries (IEEE, technology, libraries, designs)
project list in commands.do	Compile (Modelsim)	HDL projects that can be loaded and their compile order
tb_tb_tb_<library_name>.vhd	Test (Modelsim)	Multi test bench that instantiates all self-checking test benches that are available for the HDL library
unb_libraries.qip	Synthesis (Quartus)	Central list of all HDL projects

Table 5: Lists of source files for the UniBoard1 development

Table 6 shows the scripts that use some kind of find command and that are already being used for the UniBoard1 developments. In fact these scripts only search in specific sub-directories to find the project file. The disadvantage of using specific sub-directory formats is that it unnecessarily restricts the way in which HDL libraries or designs are organized in a directory.

Scripts that use find commands	**Task**	**Description**
commands.do	Compile (Modelsim)	Finds <project_name>.mpf in any /build/sim/modelsim or /build/synth/quartus/<sopc_name>_sim
unb_*	Synthesis (Quartus)	Find <design_name>.qsf in any /build/synth/quartus

Table 6: Lists of scripts that use find for the UniBoard1 development

2.3.7.1 Forward and backward lists

The lists used in UniBoard1 are considered forward list meaning that they start at the sources and typically contain all libraries that are available even if only a subset is used for the target. This can be awkward because then all libraries must compile ok even if they are not used for the target. Using backward lists implies starting at the target and then include only the libraries that it depends on. The GRLIB seems to use another approach whereby a skip list can be specified of libraries that need not be included.

2.3.8 Software build tools

2.3.8.1 Lofar software lists

For the Lofar software development CMake is used to configure (gather) the sources that are needed to build a certain target. The sources are listed in CMakeLists.txt files that are placed every directory. Together the CMakeLists.txt files at all levels in an directory tree configure the relevant sources for that directory. In fact the CMakeLists.txt file is a bit more sophisticated than merely a list of source files, because it also provides references, conditional statements and regular expressions. The ‘cmake’ command uses the CMakeList.txt files to create a parallel directory tree to keep the generated files separate from the source files tree. This parallel directory tree is then used to create the target. The target can be an executable (by means of ‘make’) but also a tar file can be a target to bundle a combination of source files and generated files for a release (by means of e.g. a python script).

The Lofar software development also uses continuous integration and reporting. This implies that for every modification in the sources the targets are rebuild and a series of predefined regression tests is done. This relies on CTest and Jenkins.

2.3.8.2 Gaisler GRLIB lists

The Gaisler GRLIB also uses list files (libs.txt, dirs.txt, vhdlsim.txt, vlogsim.txt and vhdlsyn.txt) to identify the relevant sources. The libs.txt, dirs.txt and vhdlsyn.txt are used by a global /bin/Makefile for synthesis. The vhdlsim.txt and vlogsim.txt are used for simulation only but not read by any ascii file in GRLIB. A new design will have an own Makefile that calls a Makefile for the board and the global Makefile. The Gaisler GRLIB uses ‘make’ and Makefiles as program language to provide a complete development flow from source to target.

2.3.8.3 Software build tools that are written in Python

Rather than using bash, make or cmake we would like to build the targets using Python based tools. SCons is a build system (build tool, make tool or software construction tool) written in pure Python. SCons uses Python scripts as "configuration files" for software builds. Another build system that is written in Python is Waf.

2.3.8.4 Practical requirements for VHDL build

A sofware build system ensures that:

1. all sources are build
2. only sources that changed are rebuild

The rebuild only what is needed aspect is supported for VHDL simulation, but typically not for synthesis. Incremental synthesis is possible but typically the entire synthesis needs to rerun if anything changed in a source. For simulation the simulation tool typically offers some function to create a Makefile for ‘make’, e.g. like ‘vmake’ does in Modelsim. Modelsim does not support Scons or Waf so best use ‘vmake’, Makefiles and ‘make’ to speed up compile times for simulation (similar as is done by ’mk’ in the current commands.do). For synthesis rebuild only what is needed is not yet supported, so there using Makefiles is not applicable.

For the OneClick sofware build we can use simple self-written scripts in Python (instead of in bash like in unb_* or as Makefiles in the ‘make’ language like with GRLIB) to build the software. For building the VHDL it seems not beneficial to use methods from Scons or Waf. The rebuild only what changed aspect only applies to simulation and for that the Python script can call ‘make’, because Modelsim ‘vmake’ creates a Makefile. Within the Modelsim GUI only TCL is supported, this is fine for the ‘as’ and ‘ds’ commands in commands.do. The ‘lp’ and ‘mk’ commands in commands.do can remain as TCL commands that in fact call a Python function on the command line. In this way ‘mk’ from the GUI does the same a ‘mk’ from the command line.

2.3.9 Repository directory structure requirements

The directory structure of the SVN repository needs to fulfil various requirements, such as:

1. Follow SVN common practise (trunk, branches, tags, even though we only use the trunk)
2. The structure should reflect the design flow and design philosophy
3. Multiple device support and multi tool support should be handled in one specific directory, not scattered around the tree.
4. Limited dependencies between directories, adding a feature or a variation in one directory should only have a local change and not require changes other parts of the directory tree.
5. Naming must be clear and straightforward, not necessarily short
6. Be consistent. The top level directories should not fundamentally change in time or get cluttered by extra directories
7. Hold design projects from different institutes
8. Hold different design projects (applications)
9. Hold central development environment that can be used by different projects (build scripts, one-click environment, python test environment)
10. Hold PC host application software that is used to verify and validate the designs
11. Hold PC host driver or server software that is used to monitor and control the board
12. Allow a project to define its own subdirectory structure
13. Distinguish systems (multiple FPGA), designs (one FPGA) and libraries (component)
14. Allow reuse of HDL libraries within a design project without copy-paste
15. Allow reuse of HDL libraries between design projects without copy-paste
16. Clearly separate the IP from the proprietary source code
17. Store IP or allow IP creation when needed

18. Target different FPGA families of one vendor (Stratix IV, Arria 10, …)
19. Target FPGA families of different vendors (Altera, Xilinx)
20. Target different hardware boards (Uniboard1, UniBoard^2, Roach 3, RSP, …) and systems (AUB backplane)
21. Keep board specific sources in a separate directory (pin files, mesh VHDL model, …)
22. Keep HDL library specific files together (VHDL, reference designs, IP, python peripherals, document)
23. Support using an old and a new version of a tool (e.g. Quartus 10 and Quartus 12)
24. Compiled code should be in another directory separate from the source code (e.g. /work for Modelsim or another directory outside the source tree) such that the entire build directory can be kept outside SVN and be deleted and rebuild when needed.
25. Fit the use of code development and maintenance tools like making all sources, regression tests.

3 Firmware development tools

3.1 High level tasks and tools

The development tools perform the actual tasks to get from a set of resources to a certain target, so resources → task/tool → target. Table 7 distinguishes several high level tasks:

1. **Compile** – Compile HDL source code for simulation or C code for a processor
2. **Synthesize** – Synthesize HDL source code for FPGA image
3. **Generate** – Generate HDL e.g. using a tool, existing components or a template
4. **Test** – Verify an HDL component using simulation or simulate a functional model
5. **Bundle** – Bundle a specific set of sources and target files
6. **Utility** – Provide a service (does not do a transform)

Task	Tool	Example ~	Description
Compile	Modelsim	commands.do	Compile or make all HDL
	NiosII C compiler	unb_app	Compile embedded application for the NiosII
Synthesize	Quartus	unb_qcomp	Synthesize HDL for FPGA
	OneClick stamps	unb_common.qsf	Add version and time stamps to systeminfo
Generate	DSP Builder MyHDL C2HDL OpenCL	-	Create HDL
	Megawizard	unb_mgw	Create (complex) IP
	SOPC builder	unb_sopc	Create MM board control HDL using Nios II in SOPC.
	Qsys	-	Create MM board control HDL using Nios II in Qsys.
	nios2eds	unb_app unb_bsp unb_reg unb_mif	Compile unb_osy.c for Nios II and create the register map from system.h into a systeminfo RAM initialization file.
	-	-	Create FPGA RAM initialization hex file
	quartus_cpf	unb_rbf	Create a Raw Binary File from a SOF
	OneClick MM control	-	Create MM board control interface using control master in HDL and MM bus and register map
	OneClick MM peripheral	-	Create MM register VHDL file and Python peripheral based on a register fields definition
	OneClick structure HDL	-	Create HDL hierarchy based on components connects in Python
Test	HDL simulator	tb_tb_tb_*.vhd	Regression tests using VHDL test benches
	OneClick HDL verification	-	Regression tests using the python as HVL using auto_sim
	OneClick modelling	-	Functional simulation in python of HDL components
Bundle	Web page	documents.html	Bundle relevant documentation
	OneClick release	-	Create and manage a release tag or branch
Utiltity	quartus_pgm	unb_sof	Download an sof to an FPGA
	nios2-terminal	unb_term	Open a NIOS II terminal via JTAG

Table 7: Tasks/tools that use source files to create a target

3.1.1 OneClick environment

The OneClick tool contains all scripts and programs that facilitate the generation of HDL code and the simulation of functional models. Several parts of the OneClick tool are already in use in the UniBoard1 development, but they may need some more rework. Other parts are completely new. Table 7 shows the sub tasks that can become integral part of the OneClick tool. In the UniBoard1 development the HDL compilation is done using commands.do within Modelsim and synthesis is done using unb_* bash scripts. The OneClick environment can also include makefiles or scripts (e.g. in python) for these compile and synthesis tasks. Eventually all tasks in Table 7 can get supported within the OneClick environment.

3.1.2 Tool script requirements

The scripts that perform the tasks should fulfil various requirements such as:

1. Scripts should not be too much dependent on a certain version of a tool or language.
2. Not contain source file names.
3. Not unnecessarily restrict the sub-directory structure. Use list files or find commands to configure what to do. For each resources → task/tool → target combination in Table 7 there can be need for one or more lists of files or find commands to be able to rerun the tasks automatically for all targets.
4. Not unnecessarily depend on central environment settings in a .bashrc file. Many environment settings can be made when the tool is started.
5. …

These requirements can also be applied to the current UniBoard1 workflow to adapt e.g. the commands.do and the unb_* scripts that we have for Modelsim and Quartus.

3.2 FPGA technology independent HDL

3.2.1 GRLIB two level approach

The $GRLIB/lib encapsulates technology components so that the HDL designs can be technology independent. The gencomp.vhd package provides the interface that contains generic components for:

1. Memory (block RAM, DDR2)
2. Clock buffers (PLL)
3. Pads (IO buffers)

The technology simulation models are kept in /lib/tech. We keep these files with the tool. The mapping of a generic component in gencomp.vhd on to the technology IP is done in /lib/techmap. The technologies are identified by a number in gencomp.vhd, e.g. inferred=1, altera=7, stratix3 = 26. This technology ID is passed on through the entire hierarchy of a design. The mapping from generic component to technology IP is a two level process [2]:

1. Lower level: handles the technology dependent interfacing to the specific IP cells. This lower level is implemented separately for each technology in /techmap/<technology-name>/
2. Higher level: defines a technology independent interface to the IP. This higher level is implemented only once and is common to all technologies. For each general type of IP, an entity/architecture is created at the higher level in /techmap/maps/. The entity declarations are technology independent. The architectures are used for selecting the relevant lower level component depending on the value of the tech generic.

At the lower level for each technology all generic IP components that are available in gencomp.vhd are implemented. These lower level VHDL files are technology dependent, but the technology-named-entities only use general VHDL types. Therefore at the higher level these technology-named-entities from different technologies can be instantiated together in another file that is therefore also technology independent.

Table 8 shows the files that are used to map a generic block RAM component on the Altera IP.

```
-- Technology simulation models:
tech/altera_mf/…/altera_mf.vhd            # altsyncram entity + architecture
                /altera_mf_components.vhd  # component package with altsyncram

         -- Lower level:
/techmap/altera_mf/memory_altera_mf.vhd
           # One altera_mf technology memories file with all altera_syncram_*
           # entities that fit the generic syncram_* entities and each with an
           # architecture that instantiate an altsyncram IP component from
           # altera_mf to implement the generic syncram_* for the altera_mf
           # technology.
           # VHDL: use altera_mf.altsyncram

        -- Higher level:
        /maps/all_mem.vhd    # Package with the <tech>_syncram components that
                             # are available in the memory_<tech>.vhd
             /syncram_*.vhd  # There is one file per generic syncram_* type. It
                             # contains the syncram_* entity + architecture. The
                             # architecture instantiates the appropriate
                             # <tech>_syncram from memory_<tech>.vhd for each
                             # tech generic value.
                             # VHDL: use work.gencomp.all
                             # VHDL: use work.allmem.all

        -- Higher level interface package:
        /gencomp/gencomp.vhd # package with tech number list and the generic
                             # syncram_* components for several types and
                             # sizes
```

Table 8: $GRLIB//lib example for block RAM mapping on Altera altsyncram

The GRLIB has one general purpose clkgen component that instantiates e.g. a PLL to create several clocks. Table 9 shows how the generic clkgen component maps on the technology.

```
-- Technology simulation models:
/tech/altera_mf/…/altera_mf.vhd            # altpll entity + architecture
                 /altera_mf_components.vhd  # package with altpll component

         -- Lower level:
/techmap/altera_mf/clkgen_altera_mf.vhd
             # clkgen_altera_mf entity + architecture (instantiates altpll)
             # VHDL: use techmap.gencomp.all
             # VHDL: use altera_mf.altpll

        -- Higher level:
        /maps/allclkgen.vhd  # package with all clkgen_<tech> components
                             # VHDL: use techmap.gencomp.all;
             /clkgen.vhd     # clkgen entity + architecture that instantiates
                             # clkgen_<tech> dependent on tech generic
                             # VHDL: use techmap.gencomp.all;
                             # VHDL: use techmap.allclkgen.all;

        -- Higher level interface package:
        /gencomp/gencomp.vhd # package with the generic clkgen component
```

Table 9: $GRLIB//lib example for clkgen mapping on Altera altpll

Remarks:

1. The key aspect to have technology independent IP components is the fact that the lower level entities are technology independent.
2. The GRLIB uses components for instances and component packages all_mem.vhd and allclkgen.vhd to declare all these technology specific components. If the components were instantiated as entities then the technology specific entity and architectures would need to be in kept separate files, because otherwise the technology specific library clause for the architectures of other technologies would cause compile errors because only the selected technology is supported. The advantage of directly instantiating entities is that it avoids the need for having a components package file or a component declaration. However in this case the disadvantage of instantiating entities is that all entity files need to be compiled to be able to instantiate them at the higher level. The advantage of using a lower level component package is that the lower level and the higher level can be compiled independently. The lower level technologies that are not supported do not need to be compiled.
3. The higher level component package gencomp.vhd is not needed if the technology independent IP components are instantiated as entities instead of as components.
4. The generic that selects the device technology has to be defined at the top level entity of a design and be propagated to all underlying components supporting technology specific implementations.
5. One technology can serve multiple FPGA types and one FPGA type can require multiple technologies. For example for stratixiii the clkgen uses the PLL from /tech/stratixii but the syncram from /tech/altera_mf.
6. The /techmap/inferred contains memory_inferred.vhd, mul_inferred.vhd, ddr_*_inferred.vhd. It does not contain a clkgen_inferred.vhd because a PLL cannot be inferred.

3.2.2 Technology independent HDL for UniBoard[2] and other boards

Like with the GRLIB the aim is to be able to select the FPGA technology (e.g. Stratix4, Arria10…) without having to change any source code. Multi FPGA technology support is also accommodated for in the Uniboard1 HDL development but not actually used. All FPGA dependent IP is only used via a wrapper VHDL entity file and a separate VHDL architecture file for each supported FPGA technology. For the UniBoard1 developments we use generic wrapper components for:

1. IO pads (DDIO, transceivers, 1GbE MAC, 10GbE MAC)
2. Clock buffers (PLL)
3. Memory (block RAM, DDR3)
4. FIFOs
5. Multipliers

The technology independent scheme for UniBoard1 defines one generic entity file and one or more separate technology dependent architecture files. The idea is that the technology is selected by including the appropriate technology dependent architecture file. The issue with this scheme is that it is not possible to select an architecture using a generic. Therefor it is necessary to adapt the UniBoard1 scheme. Possible solutions are:

1. The VHDL language offers a scheme of using configuration files to select an architecture for a component instance, however this scheme is difficult to manage and seems rarely used.
2. Set an environment variable that points to a technology specific directory and thus selects the appropriate technology architecture file.
3. Use a two level scheme similar as by the GRLIB [2]

The advantage of the two level scheme as in GRLIB is that the technology can be selected with a VHDL generic. Therefor this scheme is preferred also for the UniBoard technology independent HDL development. Table 10 explains how a generic memory_ram_crw_crw.vhd component can be mapped on technology specific block RAM using a g_technology generic to select the FPGA technology. The mapping from lower level to higher level is a transpose whereby at the lower level each <tech_name> directory contains a technology specific implementation of each IP component and at the higher level each IP component instantiates all technology specific components but only selects one.

```
-- Technology simulation models:
$MODEL_TECH_ALTERA_LIB/…/altera_mf.vhd     # altsyncram entity + architecture

-- Lower level:
/technology/<tech_name>/<tech_name>_<component_name>.vhd
           /virtex7
           /arria10
           /stratixiv
           /altera_mf/altera_mf_ram_crw_crw.vhd
  # The altera_mf_ram_crw_crw.vhd is name of the IP file that is generated with
  # the MegaWizard. It contains one entity + architecture. The architecture
  # merely instantiates the altsyncram component. The entity has been edited to
  # support generics that define the dimensions of the block RAM. The file is
  # technology specific because it uses the altera_mf library. The
  # altera_mf_ram_crw_crw entity is technology independent. This makes that the
  # altera_mf_ram_crw_crw component can be instantiated in a file that is also
  # read by a synthesis tool of another vendor.
  # VHDL: use altera_mf.all

-- Higher level:
technology/maps/technology_<component_name>_pkg.vhd
technology/maps/technology_ram_crw_crw_pkg.vhd
  # The implementations of an IP component for each technology are declared in a
  # technology_<component_name>_pkg.vhd similar as all_mem.vhd and allclkgen.vhd
  # in GRLIB.

technology/maps/io           # external IO: ddio, GX, 1GbE, DDR3, HMC, …
               /clock        # PLL
               /arith        # multiplier
               /fifo         # FIFOs
               /memory/      # block RAM
                      /memory_ram_crw_crw.vhd
  # The memory_ram_crw_crw.vhd instantiates all the technology specific entities
  # like altera_mf_ram_crw_crw.vhd in separate VHDL generate sections. The
  # g_technology generic determines which FPGA technology is actually selected.
  # VHDL: use technology_pkg.vhd
  # VHDL: use technology_ram_crw_crw_pkg.vhd

-- Higher level interface package:
technology/maps/technology_pkg.vhd
  # The available technologies are defined by an ID number in technology_pkg.vhd
  # similar as the tech number list in gencomp.vhd for GRLIB. The IP components
  # are not declared in the technology_pkg.vhd, because they can be instantiated
  # as entities.
```

Table 10: /RadioHDL/trunk/ example for block RAM mapping on Altera altsyncram

Remarks:

1. It is also possible to define a simulation only technology. This simulation only technology can then be used to replace technology IP by minimal behavioural models to increase simulation speed.
2. The common_ram_crw_crw.vhd in the /common library instantiates the memory_ram_crw_crw.vhd. Hence the /common library provides an extra wrapper interface that can sometimes be useful. The technology wrapping is already accomplished by memory_ram_crw_crw.vhd.
3. The VHDL files in the technology directory also follow the file naming convention to start the VHDL file name with a prefix that is the same or strongly relates to the directory name.
4. The IP in /technology/maps/io relates to external FPGA interfaces. The other IP in /clock, /arith, /fifo and /memory relates to internal FPGA resources like PLL, block RAM, etc.
5. The DDR3 IP and HMC IP have been put in the /technology/maps/io directory and not in the /memory directory, because from a digital point of view the IO aspect is the what the IP provides. The memory aspect is important from a functional point of view, but the memory itself is implemented by an external chip.

4 Description of the current UniBoard_FP7 directory structure

4.1 UniBoard_FP7 repository for Uniboard1 developments

4.1.1 Top level directories per project

The current UniBoard_FP7 SVN repository for the UniBoard1 developments fulfils already many of the requirements, but it needs restructuring to fit our future needs. It may even be necessary to start an entire new repository. The repository is called UniBoard_FP7 because the UniBoard1 project is part of RadioNET FP7. The top level directory structure defines a directory for each <project_name> and is shown in Table 15.

```
/UniBoard_FP7
    /General/trunk
    /UniBoard/trunk
             /tags/unb_release_1_0
    /Digital_receiver/trunk
    /Beacon/trunk
    /Aartfaac/trunk
    /Apertif/trunk
    /PAC
    /…
    /<Project_name>
```

Table 11: Current UniBoard_FP7 directory structure with projects as top level directories

The directory /General/trunk/ can contain information that relates to all sub projects in UniBoard_FP7. For example on how to use the SVN repository as described in this text. Whether to use one common trunk, tags and branches directory for all sub projects or products in the SVN repository or to use a trunk, tags and branches directory per sub project or product depends on how related the they are. According to http://svnbook.red-bean.com/en/1.7/svn-book.pdf:

1. If the projects are not related at all then it is best to use different repositories.
2. If the projects are slightly related then one repository with a trunk, branches and tags dir per project directory is suitable.
3. If the projects are very related then one repository with a single trunk, branches and tags dir is suitable.

The work within Uniboard FP7 fits case 2), therefore that is used above. However in retrospect case 3) applies also and would ease creating a branch or tag for projects that reuse from other project directories.

The sub projects or products in the UniBoard_FP7 repository form the top level directories, e.g.: board designs (like UniBoard, or e.g. a backplane, some IO interface card) and the various application projects of the UniBoard FP7 participants (like the EVN correlator). The top level project directories allow maximum freedom for all UniBoard FP7 participants to define their own directory structure in the UniBoard_FP7 SVN database. To get started with a new project within UniBoard_FP7 simply create a new project directory:

/UniBoard_FP7/<your-project-name>/trunk/

Typically each project or institute will master its own project directories within the /UniBoard_FP7 repository. Reading other project directories is default and will hopefully lead to reuse. Modifying other project directories is also possible, but of course this should not be done without the project owner's agreement.

Especially if you are actively developing in the UniBoard_FP7 repository, then the advice is to every few days do an SVN update at the top directory of the repository, so at:

```
  /UniBoard_FP7
```

The update time is short, so updating at the top directory is fine. The advantage is that you keep on using the latest available and it is also a way to see what is happening within the UniBoard_FP7 SVN repository.

What can occur is that due to the update, some of your code no longer compiles or runs OK, because someone else checked in a modification that affects a block that your code also uses. This is just a fact of development in a team and working on the trunk. We do not use branches, but the fact that we use different project/trunk directories avoids these issues already a lot. It is better to discover such an issue within a few days (when not so many modifications have occurred yet), then to discover many issues after months.

Typically only source files (HDL, C, Python, project files, scripts, ...) are kept in SVN. Files that are generated are not put in SVN unless they e.g. take much time to generate or should be easily available. Therefore FPGA images can also be kept in SVN, typically at the location where they are created.

4.1.2 The UniBoard/trunk project directory at $UNB

The common developments for UniBoard1 are done in the UniBoard/trunk project directory. This path is identified by an $UNB environment variable. The UniBoard project has three main top level directories /Hardware, /Firmware, /Software as shown in Table 12. For MM control interface the UniBoard1 applications use unb_osy that runs on the NiosII embedded soft core processor, so therefore there also is a /Firmware/software directory.

```
      /UniBoard/trunk/.                 # = $UNB
                     /doc
                     /Firmware/.        # FPGA HDL firmware
                              /software # Embedded software on FPGA
                     /Hardware          # Other UniBoard code
                     /Software          # PC software
               /tags/unb_release_1_0
```

Table 12: UniBoard/trunk directory structure for common UniBoard developments

Currently the $UNB directory has no branches and only one tag. This /tags/unb_release_1_0 tag was made in May 2011 to tag the test firmware release for UniBoard hardware version 2.0.

4.1.2.1 $UNB/Hardware

The $UNB/Hardware directory only contains other code that runs on the UniBoard but not on the FPGAs. It could also contain a PDF of the schematic and a board description document but is does not. The PCB development source files are also not kept in this repository. Table 13 show the contents of the .Hardware directory.

```
$UNB/Hardware/.             # Other UniBoard code
             /jtag
             /Switch
```

Table 13: $UNB/Hardware

4.1.2.2 $UNB/Firmware

Table 14 and Table 15 in more detail shows the sub levels of the $UNB/Firmware tree intended for the UniBoard firmware development. The main sections are

```
$UNB/Firmware/.                       # FPGA firmware
             /doc
             /sim                     # Modelsim setup
             /synth                   # Quartus setup
             /ip
             /modules/<library_name>  # Base libraries, IO modules
             /dsp/< library_name>     # DSP VHDL libraries
             /designs/<design_name>   # designs that map on an FPGA
             /systems/<system_name>   # multi-designs for a UniBoard, a subrack
             /software/.              # embedded C code for Nios II
```

Table 14: $UNB/Firmware top level

```
$UNB/Firmware/.                       # FPGA firmware
             /doc
             /sim/bin                 # make
                 /modelsim            # commands.do, libraries.ini,
                                      #   unb_modelsim_se.bat
                 /python              # auto_sim
             /synth/quartus           # unb_libraries.qip, unb_quartus.bat,
                                      #   user_components.ipx
             /ip                      # IP_10GETHERNET
             /modules/<library_name>  # Base libraries, IO modules
                     /common
                     /dp
                     /ddr3
                     /tse
                     /…
                     /MegaWizard/<component>   # pll, mem, arith, tse_sgmii_lvds
                     /Lofar/<library_name>
             /dsp/<library_name>       # DSP VHDL libraries
                 /fft
                 /bf
                 /…
             /designs/<design_name>   # designs that map on an FPGA
                     /unb_minimal
                     /bn_terminal_bg
                     /fn_terminal_db
                     /bn_filterbank
                     /fn_beamformer
                     /…
             /systems/<system_name>   # multi-designs for a UniBoard, a subrack
                     /aperitif_beamformer
                     /singleboard_beamformer
             /software                # embedded C code for Nios II
                      /build          # unb_* scripts
                      /modules/doc
                              /src    # module *.c/h files
```

Table 15: $UNB/Firmware details

4.1.2.3 $UNB/Software

The /Software directory contains PC host software to control the UniBoard.

```
$UNB/Software/.                       # PC software
              /cpp                     # unbctl for Apertif
              /uniserver               # server for Apertif and AAVS
              /erlang                  # MAC test scripts by Jive
              /python/.                # MAC test scripts by Astron = $UPE
                     /base
                     /apps
```

Table 16: $UNB/Software

4.1.3 Embedded software development

The embedded C code that runs on the NIOS II processor on the UniBoard itself is kept in the /Firmware directory at /Firmware/software. The C code is also setup in a modular fashion with separate c and h files per per module stored in the /modules/src/ directory and applications that include these modules and run on the Nios II stored in the apps directory. Initially there were several application programs, but nowadays only unb_osy. Hence from a firmware development point of view the Nios II with unb_osy program has become a fixed component. Any application specific control can better be done on a PC via unb_osy and the MM interface.

The Nios II with unb_osy is the MM master of the MM bus. Together the MM master and the MM bus provide remote access via UDP control protocol messages to the MM registers of an application. The main functionality of unb_osy is to acquire an IP address (using ARP) and to handle the UDP control protocol. The unb_osy functionality can also implemented in a VHDL state machine. The advantages of implementing the MM master in VHDL are:

1. In C the maximum control data rate is several Mbps whereas a HDL implementation can achieve close to the rate of the Ethernet link (i.e 1 Gbps via 1GbE or even 10 Gbps when the control goes via a 10GbE link)
2. Not using the Nios II also makes it easier to use a vendor independent MM bus. In fact the MM bus in the UniBoard application is merely a large multiplexer. This then avoids the dependency on tools like SOPC builder or Qsys.

4.1.4 Firmware development

The firmware development is kept by sub level directories like:

```
/<component_name>/doc
                 /build/sim/modelsim
                       /synth/quartus91 # Altera old
                             /quartus   # Altera current
                             /ise       # Xilinx
                 /src/ip/megawizard     # Altera
                     /ip/coregen        # Xilinx
                     /python
                     /vhdl              # RTL
                 /tb/vhdl               # Test bench
                    /python
                    /data
```

Table 17: Firmware component directory structure to target multiple platforms

The firmware component can be e.g. a:

1. **library** : consists of one component (e.g. tse, ddr3) or multiple components (e.g. common, dp, filter)
2. **design** : consists of a component that can be loaded on an FPGA (e.g. unb_minimal)
3. **system** : consists for multiple designs (e.g. apertif_beamformer)

All firmware components typically have a /sim directory for a Modelsim project file to be able to simulate and test it. A library often does not need a synthesis directory unless some trial synthesis is needed to e.g. determine the resource usage. The /src directory contains sources that are needed to create the FPGA image. The /tb directory contains sources that are needed for test. A design always needs a synthesis directory because it is intended to run on FPGA hardware. A system does not need a /src directory. By defining a /src and /tb directory it is clear which files will get synthesized into the FPGA and which not.

All HDL files in a component directory should start with a prefix that relates to the component name. In addition test bench file names have a 'tb_' prefix.

By defining a /build/sim and a /build/synth directory it is possible to allow different (versions) of HDL simulator, synthesis tools, FPGAs, boards.

The tools may sometimes enforce a certain sub directory structure, e.g. like with the SOPC generated files.

4.1.5 PC software development

The PC software development consists of:

1. Driver software for the board control (e.g. /python/base, /cpp)
2. Servers software for the board control and higher level application control (e.g. /uniserver)
3. Common software packages for test, DSP, plotting (e.g. /python/base)
4. Application test software (/python/apps)
5. …

4.2 Tools

4.2.1 Modelsim

Modelsim is installed at a central location on the machine so that it only needs to be stored once to serve multiple users. The FPGA vendor HDL IP libraries are compiled and stored in a central location as well using an IP compilation command in the synthesis tool. For each version of the PFGA vendor tool a dedicated directory is made for the IP binaries. The libraries.ini makes these IP libraries accessible for the FPGA development. Hence the Modelsim installation, the FPGA vendor IP source files and binaries for simulation are all kept outside the SVN repository.

In the Gaisler the GRLIB IP Library [2] the FPGA vendor HDL IP libraries are included in a /lib/tech/ directory. For us it seems more appropriate to keep the FPGA vendor HDL IP libraries outside the repository and rely on the IP compilation command that is provided with the synthesis tool, because then we are sure that the FPGA vendor HDL IP is up to date and complete.

For the UniBoard development Modelsim needs to be started with the project settings from commands.do, modelsim.ini and libraries.ini. For Windows Modelsim can be started using a shortcut to (a modified version of) /sim/modelsim/unb_modelsim.bat. For Linux some environment variables and an alias 'unb_sim' in the .bashrc file can be used to start Modelsim or the 'unb_msim' bash script in /Firmware/software/build.

4.2.1.1 Modelsim.ini and <library_name>.mpf

It is not necessary to overrule the default $MODEL_TECH_DIR/modelsim.ini by a project specific 'export MODELSIM=<path to project>/modelsim.ini'. Better use the default $MODEL_TECH_DIR/modelsim.ini for the Modelsim tool settings and use the *.mpf to make project specific settings. It is not possible to include the default $MODEL_TECH_DIR/modelsim.ini and then overrule parts of it. It is possible to overrule parts by making the project specific settings in the *.mpf files.

The mpf file has a [Library] section that can be used to map libraries to directory paths. The [Library] section can also have and ‘others’ clause to include more libraries that are mapped in an libraries.ini file. Unfortunately there can only be one ‘others’ clause. It is possible to chain different libraries.ini files via there ‘others’ clauses, but this is a bit awkward because this chaining enforces some fixed relation between these files. Moreover as last file in the ‘others’ chain the default modelsim.ini needs to be put to include the system libraries (like IEEE). This include reads the [Library] section of the modelsim.ini.

For UniBoard1 there is one libraries.ini that is manually maintained and includes all available HDL libraries. The GRLIB takes another approach whereby except for the system libraries all other libraries are compiled into the default work library. This scheme avoids the need of a libraries.ini file, because is only needs to map work to the work directory. The minor disadvantages are that for each target all required libraries are compiled again and that components with the same entity name in different libraries cannot exist.

For UniBoard1 the <library_name>.mpf is maintained manually using the GUI and an editor. The GRLIB takes another approach whereby a vhdlsim.txt file lists all sources and the Makefile creates the mpf for it. For UniBoard2 we can take a similar approach whereby the mpf is created from a list file. The advantage is that the list file is tool independent and easier to maintain. The Modelsim mpf file has three sections [Library], [Project] and [vsim] as shown in Table 18. These sections can be filled by a script using the list file and some template settings. There are some more settings but these are not essential.

```
[Library]
others = $UNB/Firmware/sim/modelsim/libraries.ini
[Project]
Project_Version = 6               ; must be > 5
Project_DefaultLib = work
Project_SortMethod = unused
Project_Files_Count = 217
Project_File_0 = ../../../src/vhdl/common_complex_mult_add_pipeline.vhd
Project_File_P_0 = … folder src … compile_to work compile_order 101 …
…
Project_File_216 = ../../../src/vhdl/common_blockreg.vhd
Project_File_P_216 = … folder src … compile_to work compile_order 112 …
Project_Sim_Count = 57
Project_Sim_0 = tb_common_multiplexer
Project_Sim_P_0 = Generics {} timing default … work.tb_common_multiplexer …
…
Project_Folder_Count = 4
Project_Folder_0 = src
Project_Folder_P_0 = folder {Top Level}
Project_Folder_1 = ip
Project_Folder_P_1 = folder {Top Level}
Project_Folder_2 = tb
Project_Folder_P_2 = folder {Top Level}
Project_Folder_3 = tb_tb
Project_Folder_P_3 = folder {Top Level}
…
[vsim]
RunLength = 0 ps
resolution = 1ps
IterationLimit = 100
DefaultRadix = decimal
```

Table 18 Modelsim project file (mpf) example

4.2.1.2 Modelsim commands.do

The UniBoard1 commands.do constains useful vsim commands that can be used within the Modelsim GUI:

• 'as'	Add signals hierarchically to Wave window
• 'ds'	Delete all signals from Wave window
• 'lp'	Load HDL library project file
• 'mk'	Make HDL library project file(s)

The commands.do also contains the list of available design and module library projects. This list needs to be maintained manually. The commands.do is included by means of an environment variable via 'export MODELSIM_TCL=$UNB/Firmware/sim/modelsim/modelsim.tcl'. However it is preferred not to use the MODELSIM_TCL because it is only still supported for backwards compatibility of Modelsim. Instead commands.do can be called via the vsim -do command line argument option.

The commands.do is a TCL file, because Modelsim uses TCL. The ‘as’ and ‘ds’ commands can remain as TCL commands, but the ‘lp’ and ‘mk’ commands could be replaced to call a Python script on the command line that does the actual work. In this way all HDL build knowledge can be kept in Python.

4.2.2 Quartus

Quartus is installed at a central location on the machine so that it only needs to be stored once to serve multiple users.

For UniBoard development Quartus needs to be started in Windows using (a modified version of) /synth/quartus/unb_quartus.bat. The start script set $UNB and calls Quartus. On linux the $UNB is already set in the bashrc file so then Quartus can be called directly via ‘quartus –64bit &’.

4.2.3 Tool start script

Using a script (*.bat on Windows or *.sh on Linux) is preferred because then the environment variables can also be set in that script (instead of in the .bashrc). This allows e.g. starting two versions of Modelsim in parallel by using a dedicated start script for each tool version. Some preliminary new scripts show that it is possible to start the tool GUI or an unb_* script avoid having central environment settings in the .bashrc:

```
$UNB/Firmware/software/buil/modelsim_version.sh
                           /quartus_version.sh
                           /quartus_generic.sh
                           /project_settings.sh
```

4.3 List of inconveniences and other remarks

1. The /UniBoard name refers to a board, but it contains firmware that could be reused on other boards as well
2. The python environment is kept in /UniBoard/Software/python, but is used by other projects as well.
3. HDL libraries are kept in /modules but some are also kept in other locations in /dsp and in /modules/Lofar.
4. The sim and synth directory levels in /build/sim/modelsim and /build/synth/quartus can be omitted to reduce the directory hierarchy, because the tool directory name already suggests the build purpose.
5. The IP is kept in different places. Some IP is kept in the central /ip and /modules/Megawizard directories and some IP is kept in a local src/ip at in the <library_name> directory.
6. Reference designs could perhaps better be kept with the <library_name> directory to have it near the module that it shows.
7. The /systems directory forms a structure for a combination of designs and is placed in parallel to the /designs directory. If a design is not reused then it could also be placed within a <system_name>

directory. If a design can be reused in different systems, then it seems fine to place that design in the /designs directory.

8. The documents in the Firmware/doc directory could probably be moved to $UNB/doc to have one central /doc and only local <component_name>/doc per library, design or system
9. The node_* component of designs is reused, which is nice but also a bit strange
10. The unb_* scripts are kept in /Firmware/software/build which is too deep and not logical because the /Firmware/software directory is intended for embedded software. A more suitable location for the unb_* scripts would have been the /build directory.
11. The auto_sim.py script calls Modelsim vsim and could therefore be kept in sim/modelsim. It is not needed to define a sim/python directory for it because the script language itself is not essential.
12. The user_components.ipx file that makes UniBoard specific port usable in SOPC builder is kept in SVN but needs to be copied to the Quartus install directory.
13. Preferably generalize the way in which a tool is started to avoid discrepancies.
14. The /UniBoard directory also contains application designs for Apertif

ASTRON

5 Proposed SVN directory structure for UniBoard^2

5.1 Where to put the trunk, branches and tags directories?

For UniBoard1 developments we have once made a tag (unb_release_1_0) and otherwise done everything on the trunk. The trunk, branches and tags introduces an extra level of hierarchy in the directory paths but it is wise to keep it even though we have not used it much so far. The trunk can start ether at:

1. at the top of the repository, so all on one trunk
2. one or even two directories lower, this then will yield many lower level trunks

For the UniBoard_FP7 repository scheme 2 was used to allow each application to have its own branches and tags. However to ease reuse between the applications it is better use scheme 1 because then a branch or tag will automatically also cover any dependencies between applications. Furthermore scheme 1 is the simplest and therefore preferred for new UniBoard^2 developments.

5.2 Work in UniBoard_FP7 or start a new RadioHDL repository?

The options are:

1. Evolve the current UniBoard_FP7 repository into the new repository structure
2. Start a new RadioHDL repository and copy over what is needed for UniBoard^2
3. Use an SVN branch to continue the UniBoard1 developments
4. Use SVN externals to preserve the link between UniBoard1 sources for UniBoard^2
5. …

The advantage of option 1 is that it preserves the version relations, but the disadvantage is that it may be too difficult to manage using the old Uniboard1 approach and a new development approach for UniBoard^2. Furthermore having two development approaches in on repository may be confusing. Option 2 has the advantage that it means a clean start. However can we know in advance whether the new directory setup is actually sustainable? Option 3 requires changing the path for the uniBoard1 developments and both option 3 and 4 rely on SVN features. An intermediate approach of option 1 is to start the developments for Uniboard^2 on a top level /trunk in the current UniBoard_FP7 repository as shown in Table 19. In this way the UniBoard1 approach is preserved and the new UniBoard^2 approach is on the new top level /trunk conform the all on one trunk approach.

```
/UniBoard_FP7
    /<project_name>/trunk
    /UniBoard/trunk
    /EVN_Correlator/trunk
    /Aartfaac/trunk
    /…
    /<RadioHDL tree>/  # top of RadioHDL directory structure for uniBoard²
```

Table 19: Start new directory structure for RadioHDL as a project in UniBoard_FP7

The directory structure of Table 19 makes it feasible to gradually go from the UniBoard1 development approach where each project has its own trunk to the new RadioHDL approach. Both approaches can co-exist and do not confuse each other because the new directory for the RadioHDL is entered via the single top level trunk. A slight inconvenience is that the repository name UniBoard_FP7 is a bit out-dated. A disadvantage of defining the /RadioHDL tree within the /UniBoard_FP7 repository is that there may grow dependencies between the old and the new way of working that are difficult to manage and that even could cause existing UniBoard1 designs to fail. Therefore it is considered better to start /RadioHDL as a new repository. The /UniBoard_FP7 repository then can remain as it is and no sources are effected by moving to the new development approach of /RadioHDL.

5.3 RadioHDL repository

5.3.1 Top level overview

Table 20 shows the top level directory structure for the new /RadioHDL repository. The top level directories contain HDL ‘libraries’ that can be mapped on a certain FPGA ‘technology’ to make an ‘application’ for a ‘board’ that uses these FPGAs. To do this application development the directory contains ‘tools’ and other ‘software’. The top level directory structure resembles the GRLIB top level in Table 4.

```
/RadioHDL                           # project in /UniBoard_FP7
  /tags
  /branches
  /trunk
      /libraries/<lib_name>         # HDL libraries and technologies for reuse
      /software/<program_name>      # Various other software
      /tools/<tool_name>            # Development tools
      /boards/<board_name>          # FPGA boards
      /applications/<project_name>  # Applications for the boards
```

Table 20: New RadioHDL repository directory structure top level

5.3.2 Sub directories

Table 21 shows the more details of the /RadioHDL directory.

```
/RadioHDL                         # as <project_name> in /UniBoard_FP7
  /tags
  /branches
  /trunk/
      /libraries/<lib_name>
                /external         # fmf, easics, numonyx
                /base             # common, dp, diag, uth, reorder, …
                /dsp              # pfb, bf
                /io               # tse, tr_10gbe, ddr3, i2c, flash
                /composite        # hierarchical library of base, dsp, io, …
                /misc
                /technology       # stratixiv, arria10, virtex4, …
      /software/<software_name>   # various other software
      /tools/<tool_name>
            /quartus              # synthesis build scripts
            /modelsim             # simulation build scripts
            /oneclick             # various Python scripts
      /boards/<board_name>
             /uniboard1/designs/unb_common
                                 /unb_minimal
                       /software/nios2/apps      # unb_osy
                                        /modules
                                  /erlang
             /uniboard2/designs/unb2_minimal
      /applications/<project_name>/designs
                                  /systems
                                  /software
```

Table 21: New RadioHDL directory details

All applications can be put in the /applications directory. In theory an application could run on any technology and board, however in practise an application will often be board specific. An application that is quite board specific like a board test image can be put in the /boards directory. Therefore the /boards and /applications directories both have sub directories like /designs.

The /boards directory does not contain board design file. It only contains what is needed from a firmware point of view. The /designs/<board>_common design can contain e.g. the pinning files (tcl), constraint files (sdc), behavioural HDL models of board IO. This directory could also contain the board control module in case the technology independent composite board control component cannot be used.

The /firmware directory level has been skipped. Both embedded software and PC host software can be kept in the same /software directory. Host software can be general purpose, or board specific or application specific, therefor the /boards and /application directories also have a /software sub-directory. Where to put for example new software depends on the expected or possible reuse:

1. In **/software**, if it is clear from the start that it can be reused
2. In **/boards/<board_name>/software**, if the software is initially board specific like the existing python and erlang code for Uniboard1
3. In **/applications/<project_name>/software**, if the software is application specific

It is also possible to reused sources between lower level directories, e.g. directly reuse (and depend on) sources from /uniboard1 in /uniboard[2]. However it may become less clear that there are reuse dependencies and therefore better create a generic library for it then or to make an SVN copy into the local board directory if reuse is too difficult to manage.

Table 21 also shows the /boards/unboard1. This directory is not strictly needed, because the UniBoard1 developments are already kept as a top level directory in the UniBoard_FP7 repository (see Table 11). However the /boards/unboard1 directory can be used port existing UniBoard1 applications and for new UniBoard1 applications, especially if they rely on the OneClick tools.

5.3.3 Firmware subdirectory structure

In the GRLIB the lists of files for simulation and synthesis are tool independent and kept at the top level of the directory. This scheme implies that the entire build directory for simulation or for synthesis can be created and kept outside SVN. With UniBoard1 the build directory does contain the mpf and qsf project files. The build directory can be deleted, but then an SVN update is needed to recover this sources,

Table 22 shows the subdirectory structure for an HDL component. It is the same as Table 17 for UniBoard1 except that the /sim and /synth directory levels in /build have been skipped. In this scheme the build compile results for modelsim are still kept at the library, which is fine. For synthesis the build directory contains a list of files (library_name.qip) and for designs it also contains the synthesis result. With Modelsim it is convenient to keep any wave.do files within the mpf directory, because then they are most easily loaded from the GUI. For UniBoard1 using SOPC Builder implies that the SOPC description file needs to be kept with the qsf file, because otherwise paths in the generated SOPC qip file are not correct. For UniBoard1 the mpf of designs needed to be kept in the /quartus/sopc_<design_name_sim/ directory because otherwise fixed paths in the generated VHDL files are not correct. However typically we do not simulate the SOPC files anymore, because in simulation the MM files model is sufficient. Therefore this awkward placement of the mpf is no longer needed. For UniBoard1 and UniBoard2 in the new /RadioHDL tree we can support QSys and skip the support for SOPC. The option to have the build directory outside SVN for both synthesis and simulation is nice to have for /RadioHDL but not essential, because with UniBoard1 it has not been an issue.

```
/<library_name>/build/modelsim/work
                     /quartus/
               /doc
               /src/vhdl
                   /py
                   /data
               /tb/vhdl
                  /py
```

Table 22: Directory details for an HDL component (can be a library or a design)

5.3.4 Build scripts

The build scripts are may be tool independent or not. Initially it seems easiest to have scripts dedicated per tool. Later on another script could then call the tool dependent scripts to make a tool independent script if it becomes necessary to support multiple synthesis tools.

The advantage of using build scripts that depend on generic configuration files to create tool specific project files is that it becomes more clear what settings are made. Having a generic configuration to build a target makes it easier to automate the tool flow. The disadvantage is that it may be difficult to create a tool specific project file and to define a limited set of parameters. However the GRLIB Makefile shows that it is feasible for many simulators, synthesis tools and FPGA technologies. For Quartus the UniBoard1 development also shows that it is possible to use a limited set of configuration parameters for all our designs (e.g. unb_common.qsf).

The build scripts can depend on txt files with lists of sources. The advantage of using plain lists like with the GRLIB is that they are simple to use. The advantage of lists that support control like with CMake is that they can steer the build. An intermediate form is to define the HDL library info as a dictionary of key words + items. This allows using a single hdl_lib_info.txt file to define all build parameters for compile, verification, validation and synthesis. The advantage of using a dictionary format is that the file is still simple because it does not have program statements (like if, for, function calls), but still provides some flexibility by adding extra key parameters and avoids the need to define a separate list file for each different target.

Suppose every library and design directory has a **hdl_lib_info.txt** file. By searching for all hdl_lib_info.txt files in the /RadioHDL tree a script can find all available HDL libraries.

For simulation it is feasible to generate a mpf from based on the name of the library, the path to the /work directory and a list of the libraries that it directly depends on. Using the depend libraries the script can recursively find all HDL libraries that are required for this target and map these in the mpf. Alternatively the hdl_lib_info.txt may also contain a key with all libraries that are required to avoid having to construct these. Another key in the hdl_lib_info.txt file contains a list of all HDL source files (typically from /src and from /tb) that are relevant for simulation in compilation order. For all test bench files the script should create a simulation configuration statement in the mpf. The test bench files do not need to be marked explicitly because it is fine to recognize them by the 'tb_' prefix in the file name. However the hdl_lib_info.txt could also contain a key that identifies all test bench files that need a simulation configuration statement in the mpf.

The 'lp' command and 'mk' command assume that the mpf of the HDL libraries exists, because they operate using the mpf files, similar as with the commands.do for UniBoard1. The 'lp' command sets the current target library and loads its mpf. This can be a design project but also a library project. The available libraries are found via the hdl_lib_info.txt files. The 'mk all' command makes all libraries that are needed to compile the current target library. Using 'mk' only makes the current library. If the /work directory does not exist yet then it is first created. Using 'mk clean' deletes the current /work library and 'mk clean all' deletes all /work directories. Use 'mk compile' to only compile the project and without creating a Makefile.

For regression tests using only VHDL test benches the hdl_lib_info.txt contains a key with a list of all tb that need to be run. These test benches have to be self-checking so that they report a 'passed' or 'failed' result and the have to finish when done, so they can be ran with 'run -all'.

For regression tests using Python tests cases via the MM control interface the hdl_lib_info.txt contains a key with all tc - tb that need to be run with Modelsim. Similar another key can identify all tc that need to run for validation on hardware with a target board.

For synthesis a key contains the top level entity file and all the HDL source files (typically from /src) that are relevant. The hdl_lib_info.txt could also contains keys to support multiple revisions of a design, each with another set of generics. This would avoid the need to rely on the synthesis tool to support revisions. It is not clear yet how to include e.g. qip files that are generated by e.g. QSys.

5.3.5 Board control components

The board control component provide MM access via UDP and can be kept at:

1. /libraries/composite/ : For a generic board control component. Parts of it are generated like the dimensions of the MM bus.
2. /boards<board_name> : For a board control component that is board specific.

5.3.6 Host software

1. /software : general software can be kept in the most central software directory
2. /applications/<project_name>/software : for project specific software
3. /tools/<tool_name> : for generic software that belongs to a tool environment, e.g. the various packages in /python/base

Driver or server software can be kept in the central /software directory. If it is more application specific then the software can be kept at /applications/<application_name>/software. E.g. the uniserver for Apertif could have been put at /applications/apertif/software/uniserver.

The Python package that we have for DSP data analysis could be left in their current place or SVN copied and further maintained in /tools/oneclick/base/.

5.3.7 Embedded software

The Nios II software that is available for UniBoard1 could also be reused for other boards that have a Nios II softcore. However the only purpose of the Nios II software that has remained for Uniboard1 is unb_osy. In fact the Nios II with unb_osy can be considered as a component that is similar to other HDL modules. It is possible to reuse code between project, so it is not strictly necessary to keep reused code as central as possible.

The PowerPC on Roach2 runs Linux and thus makes the distinction between embedded software and host software less sharp. The Roach2 software for the PowerPC is named Borph.

The embedded software like unb_osy for the Nios II and Borph for the PowerPC can be kept at the central /software directory or at a local board directory in /boards/<board_name>/software. Using the central location more clearly shows that the embedded software may be reused by multiple boards. Using a local directory avoids cluttering the central software directory with board specific software. The local directory still allows reuse between boards.

6 Conclusion

Moving towards the /RadioHDL directory tree is needed to ease future developments where we need to:

1. support multiple technologies (e.g. startix4 and arria10)
2. support multiple tools (e.g. quartus 11 and 12)
3. have higher level or faster design (OneClick functional modelling and code generation, ease reuse)
4. have continuous integration testing
5. …

The movement can be done gradually, by maintaining and improving the current Uniboard1 development tree and filling in the new directory structure for /RadioHDL when needed:

1. UniBoard1 developments can continue at their current location.
2. UniBoard^2 designs can start in a new /RadioHDL repository. The first design is a pin check design that is needed to ensure the schematic pin assignments have no conflicts.
3. The HDL libraries are copied to the /RadioHDL repository and made technology independent.
4. Existing UniBoard1 designs that are still relevant are also ported to the /RadioHDL repository and reproduced there, starting with unb_minimal.
5. The UniBoard1 repository serves as back up and as main repository for designs that have not yet been ported.
6. New UniBoard1 designs preferably start in the new /RadioHDL repository.
7. In /RadioHDL SOPC builder is no longer supported, instead QSys is used. Later on QSys can be phased out when a generic MM control master and bus can be generated based on a MM register map.
8. New OneClick functionality can start in the /RadioHDL/trunk tree.

This approach implies that for some time both the UniBoard_FP7 repository and the RadioHDL repository will be actively used. Once our main applications are have been reproduced in the /RadioHDL repository then the /UniBoard_FP7 repository will become less relevant. For institutes that only develop for UniBoard1 the /UniBoard_FP7 repository can remain the primary repository. The transition time to port all relevant libraries and designs from UniBoard1 to the /RadioHDL repository depends on when it is necessary to reproduce a working design in the new build flow. During the transition time there will be two copies of the reused HDL libraries, e.g. common library in /UniBoard_FP7 and a common library in /RadioHDL.