Version control friendly project management system for FPGA designs

Wojciech M. Zabołotny[a]

[a]Institute of Electronic Systems, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland

ABSTRACT

In complex FPGA designs, usage of version control system is a necessity. It is especially important in the case of designs developed by many developers or even by many teams. The standard development mode, however, offered by most FPGA vendors is the GUI based project mode. It is very convenient for a single developer, who can easily experiment with project settings, browse and modify the sources hierarchy, compile and test the design. Unfortunately, the project configuration is stored in files which are not suited for use with Version Control System (VCS). Another important problem in big FPGA designs is reuse of IP cores. Even though there are standard solutions like IEEE 1685-2014, they suffer from some limitations particularly significant for complex systems (e.g. only simple types are allowed for IP-core ports, it is not possible to use parametrized instances of IP-cores). Additionally, the overhead associated with packaging of IP-cores is significant and not justified for simple reusable blocks.

This paper presents a system aimed at storing the whole design in a VCS oriented form. The hierarchy of sources is described with textual "extended project (EPRJ) files" which are fully controlled by the user and may also be put in a VCS. The IP blocks may be easily added to the project just by including the accompanying EPRJ file. Both absolute and relative file paths may be used which allows the flexible structure of directories. The sources of locally developed IP blocks may be stored in directories located inside the main source tree, while sources of independently developed blocks, using separate VCS repositories, may be located outside that tree. The environment allows splitting the design into smaller parts, which are synthesized independently. That reduces the time needed to recompile the whole design if only a few blocks are modified. The system creates the standard project, which can be used for convenient interactive work with the design. After the interactive session, the user should transfer changes of settings into the system files (also under VCS control). With that approach, it is always possible to recreate any stable version of the project from the VCS. The system also provides a possibility of automated rebuilding of the design from the VCS stored files. That is especially useful for "build servers" used in serious projects. The development of the system was inspired by the needs of firmware development for the CBM experiment. The system has been developed mainly for Xilinx Vivado tools, but adaptation for Altera Quartus is planned in the nearest future. The system is developed as a free and Open Source solution.

Keywords: FPGA, project management, version control, build system, Vivado, Xilinx

1. INTRODUCTION

Modern FPGA design tools are usually oriented on interactive work with Graphical Using Interface (GUI) in so-called Project Mode. This mode of operation is very convenient for the developer, who can interactively modify the settings used to build the firmware. It is especially useful for beginners who may easily discover and test available features, navigating through hierarchical menus. For advanced users, there are sophisticated Tcl commands available.[1–3]

The whole HDL project consists of the HDL sources, constraints describing e.g. the I/O pin assignment, location of functional blocks and timing requirements, synthesis and implementation settings. During the development and debugging of the design, all those components may be modified, and it is important, that the user may record the history of changes to discover what a particular change has cured the certain problem or when another problem was introduced. To fulfill that requirement, it is essential, that all changes may be traced by the Version Control System (VCS). There are many such systems available, like Subversion,[4] Git,[5] Mercurial,[6] Bazaar[7] and many others. Even though most of those systems are dedicated to software development, they can be used for HDL projects as well. There are however some problems, which must be solved.

Further author information: (Send correspondence to W.M.Z.)
W.M.Z.: E-mail: wzab@ise.pw.edu.pl, Telephone: +48 22 234 7717

1.1 Incompatibilities between FPGA development environments and VCS systems

Even though the HDL sources may be treated as normal source files and can be easily controlled via a VCS, other parts of the HDL design are usually stored in a way which is difficult to store in a VCS. For example, in Xilinx Vivado[8] environment, the hierarchy of HDL sources and constraints together with the synthesis and implementation settings are stored in an XML file. Even though this file stores the complete information about the project organization; it is difficult to obtain the pure list of changes introduced to the project, by a simple comparison of two versions of that file. Therefore, the simple putting the project file into the VCS is not a viable option. Vendors of FPGA chips are aware of that situation. They publish some recipes describing suggested ways of using their tools together with the VCS systems,[9–12] however those solutions are inconvenient to use.

1.2 Problem with hierarchical designs with IP blocks

A very important aspect of HDL design is the possibility to reuse the Intellectual Property (IP) blocks as components. There are standard vendor-specific methods to handle them, for example, for developers of firmware for Xilinx FPGAs; there is a dedicated IP packager tool.[13,14] There is even an IEEE standard describing how to package them in a portable way.[15] Unfortunately, these standard approaches do not support a functionality, that is very important in complex designs - the possibility to use complex types in ports of IP blocks. For example in VHDL-based designs, this functionality allows connecting a complex interface using just two records - one for input signals and the another for output signals. An example of such approach may be the IPbus[16] interface used in firmware developed for High Energy Physics (HEP) experiments.

Using of IP cores packaged for the particular tool may also be a source of problems if we need to have a portable block, that can be used in FPGAs manufactured by different vendors. It may be necessary to package them separately for each brand of FPGA.

It is also problematic how to handle independently developed IP cores which are maintained in separate repositories.

2. ORIGIN OF THE SYSTEM

The development of the described system was triggered by necessity to develop the firmware for CBM[17] experiment, based on the newest Xilinx FPGAs. The final firmware must integrate IP blocks developed by different groups, and efficient mechanisms for development, verification, and continuous integration must be provided. The CBM team uses the mature system (developed by D.Hutter) based on makefiles and scripts for automatic compilation of the firmware from sources, with the possibility to use sources managed by the VCS. However, that system was developed for the old Xilinx ISE environment. Unfortunately, that system is an internally used solution, and its details have not been published. The necessity to adapt that system to the new Vivado environment gave the impulse to attempt to create a fresh solution aimed specifically on the new design flow.

Another interesting existing solution is the build system developed by W.F.J.Müller in his "w11" project.[18] This system uses the hierarchically organized description files (VBOM - "VHDL bill of materials"). The main component of that system is the *vbomconv* script, which uses the VBOM files to create the necessary makefiles, project definition files and simulation files. The *vbomconv* script is written in Perl language and is quite complicated, as it includes the logic necessary to generate all mentioned types of output files.

The developed system should be as simple as possible. Its idea is based on the old textual "prj" project files used in old versions of Xilinx ISE. Therefore the system is named VEXTPROJ (**V**ivado **ext**ended **proj**ect). If in the future also other brands of FPGAs will be supported, the name may be explained as "**V**ersatile **ext**ended **proj**ect".

To ensure low complexity, the VEXTPROJ relies on features provided by the target development environment wherever possible. Therefore the Vivado backend is implemented in Tcl, which is the native language for Vivado extensions. It also fully relies on Vivado mechanisms regarding the analysis of sources dependencies. That allowed relatively simple implementation, which should be easy to adapt to the newer versions of Vivado.

The operation of Vivado VEXTPROJ backend is based on the concept described in [19], and assumes that the Vivado project is only an ephemeral entity, created from the project description files and sources. After creation, it may be used to compile the firmware and/or to perform some development interactively, but finally, all the modifications should by stored in the sources and project description files, and afterward the project may be deleted.

3. IMPLEMENTATION OF THE VEXTPROJ

The VEXTPROJ in its initial form aimed at the creation of Vivado projects, originated from the scripts generated by the *write_project_tcl* command available in Vivado. Those scripts contain both commands used to create the project and the data, describing the sources, constraints and other files used in the project. In VEXTPROJ, the commands used to create the project have been grouped in the *eprj_build.tcl* file, while the data describing the files are moved to the dedicated EPRJ files. In future versions of the VEXTPROJ, those files should be portable. The same EPRJ files should be able to describe the design for tools provided by different vendors*. Currently, the Vivado backend for the VEXTPROJ consists of four Tcl scripts:

proj_def.tcl Defines a few fundamental constants for the created Vivado project (e.g., the project name, the FPGA type, location of the top EPRJ file and others). This script is not executed itself. It is executed by the next scripts.

eprj_create.tcl Creates the Vivado project using values defined in the *proj_def.tcl* file starting from the top EPRJ file.

eprj_write.tcl Writes the initial state of Vivado project created by the previous file to the *initial_state.tcl* file (see section 5 for details).

eprj_build.tcl Builds the project created by the *eprj_create.tcl* script.

The last three scripts may be called automatically in order from the BASH script:

```
#!/bin/bash
set -e
vivado -mode batch -source eprj_create.tcl
vivado -mode batch -source eprj_write.tcl
vivado -mode batch -source eprj_build.tcl
```

Of course, they may also be called individually from the command line. The status code returned by the script, or by vivado may be used to check if the particular step of processing was completed without errors.

4. SYNTAX OF THE EPRJ FILES AND RELATED FUNCTIONALITIES

The EPRJ files are simple text files consisting of lines describing the HDL project. The first word in the line is always a command, defining the type of the line and its meaning. All currently available commands are listed and shortly described in the next subsections.

4.1 Definition of files used by the project

The main purpose of the EPRJ files is to provide the list of the files (sources, constraints, and others) used by the design. Those files may be defined using the following lines:

vhdl library_name file_name Adds the VHDL source to the project.

xci library_name file_name Adds the IP block packaged in the XCI file to the project.

xcix library_name file_name Adds the IP block packaged in the XCIX file to the project.

header file_name Adds the Verilog header file to the project.

global_header file_name Adds the global Verilog header to the project.

sys_verilog file_name Adds the System Verilog source to the project.

verilog file_name Adds the Verilog source to the project.

*Of course, some details are hardware specific, and therefore additional conditional commands will be added in the future versions of EPRJ format

bd file_name Adds the Block Design component to the project.

mif file_name Adds the MIF file to the project.

For certain source files, the *library_name* field is available, which specifies the library, the source belongs to.

The next two lines describe the constraints:

xdc file_name Adds the XDC constraints file to the project.

xdc_ooc file_name Adds the Out-of-context XDC constraints file to the project (only to the blocks selected for OOC synthesis - see section 4.3 for details).

The files added to the project by the earlier-described lines may be given certain properties with the next two commands:

prop property_name property_value Sets the particular property of the last source or constraints file to the given value).

propadd property_name property_value Adds the given value to the list of values of the particular property of the last source or constraints file).

It is possible to use multiple *prop* or *propadd* lines in sequence below the source or constraints file to set multiple properties, or to add multiple values to the same property.

4.2 Building the project hierarchy

The above commands allow defining the simple project with all components defined in one EPRJ file. However, usually, the reasonably complex designs have a hierarchical structure. The groups of source files build the IP blocks which are later on instantiated by the higher level block. To facilitate that structure, the EPRJ format allows creating the hierarchy of EPRJ files. The developer may create the EPRJ file describing certain IP block, and then include it into the upper-level EPRJ file. The appropriate line is:

include path_to_directory_or_eprj

If the path to the directory is provided, then the file *main.eprj* in that directory is searched for and included. If the path to the file is provided, the pointed file is included. The important fact is that the *include* command accepts both, the relative and the absolute paths. The relative paths are appropriate for locally defined IP blocks, while the absolute paths allow to create the directories containing the whole libraries of IP blocks, and include them in different projects. The example of such hierarchical project is shown in Figure 1.

4.3 Using the Out-of context (OOC) synthesis

In fairly complicated designs, it is important that after some modifications only the changed parts are resynthesized. In Vivado this can be achieved by marking particular blocks for Out-of-context (OOC) synthesis. In the EPRJ format it can be achieved using the following line:

ooc stub file_name blk_top_entity

The *stub* field may be set either to "auto" (meaning that Vivado should automatically create the stub[†] file), or to "noauto" (meaning that the stub file is provided by the user). Additionally, it is necessary to provide the name of the EPRJ file describing the IP block selected for the OOC synthesis and the name of the top entity in this block. The possibility to use the user-provided stub file is essential to allow the OOC synthesis of blocks using the I/O ports with complex types. As it was mentioned in the introduction, use of such ports may significantly simplify complex designs. Unfortunately, up to now Vivado is not able to generate correct stubs for such blocks – it simply ignores all the ports with complex types.

[†]The stub file is an HDL file describing only the I/O ports of the block. It is used to assure proper synthesis of the block containing the OOC selected block. In the implementation phase, the stub is replaced with the netlist generated during the OOC synthesis.

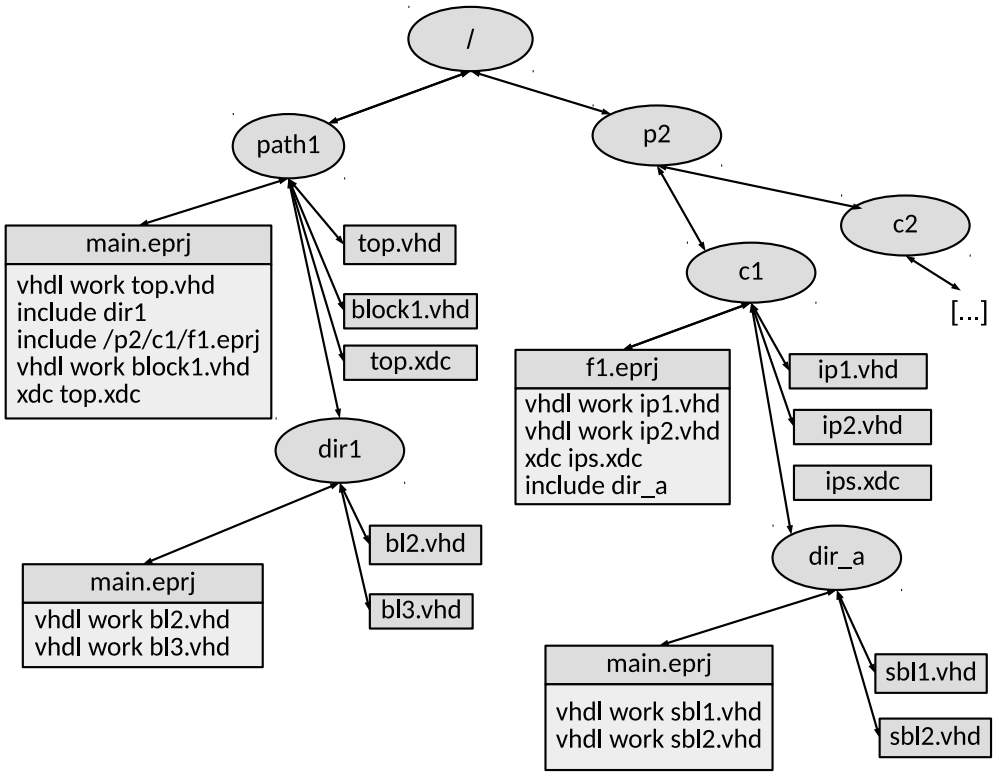

Figure 1: Example of the hierarchical project. The building of the project is started in directory */path1*. The top EPRJ file is *main.eprj*. This file adds 2 VHDL files, one XDC file and includes two folders containing definitions of other IP blocks. In the case of the local IP block (*dir1*), the relative path is used. The directory */p2* contains the library of IP blocks, including the block *c1* used in that design. The absolute path is used to include the IP block defined in the */p2/c1* directory. As this block is described using the EPRJ file with the non-standard name (*f1.eprj*), the full file-path is specified. The IP block *c1* also contains a nested IP block, defined locally in the directory *dir_a*. The order of lines in the EPRJ file may vary (however in some cases, e.g. when using the *prop* or *propadd* lines it may be important).

4.4 Working with remote repositories

As it was mentioned in the introduction, when the design is created from blocks developed and maintained independently by different teams, it is important that the sources used to build the firmware may by downloaded from different VCS repositories. It is also important that the version of sources used is strictly defined. In VEXTPROJ this is achieved using one of following commands:

git_remote repository_url tag_name [exported_directory [stripped_comp_num]]

git_local clone_directory commit_or_tag [exported_directory [stripped_comp_num]]

svn repository_url_with_exported_path [revision]

The first of those lines allows downloading sources from the remote GIT server. For example:
git_remote git://my_server.com/repo1 v1.0
Such a line downloads the version of sources tagged with "v1.0" of sources kept in the GIT repository with URL *git://my_server.com/repo1*[‡]. The downloaded sources are put into (newly created or purged) subdirectory *ext_src*.

[‡]If the *tag_name* field is omitted, then the newest version of sources will be used. Of course, that option should not be use to generate the production quality firmware

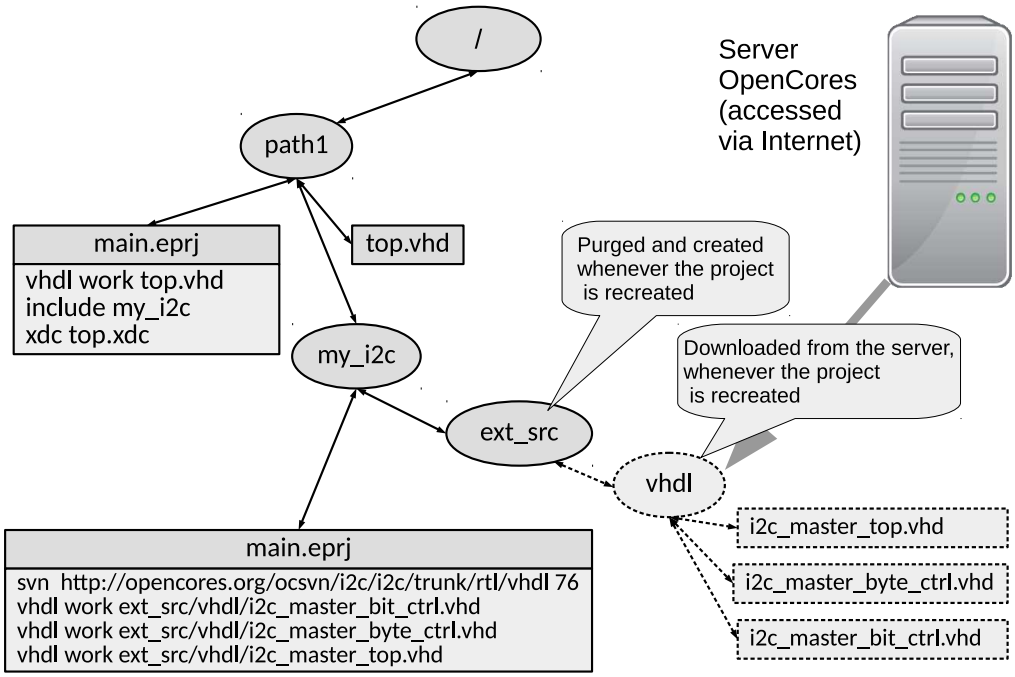

Figure 2: Example of the project, which imports the version 76 of the I2C controller sources available in the OpenCores SVN repository. (You must ensure, that the user running the Vivado can log in automatically to the OpenCores SVN server). This example shows, how to handle the imported sources (not compatible with VEXTPROJ) with an additional EPRJ file located outside the imported directory.

However sometimes the remote repository may contain a huge library of IP blocks, and we are interested only in one of them. In this case, we may tell the VEXTPROJ to import only a certain subdirectory from the repository. If the path to the required subdirectory is long, we may instruct the VEXTPROJ to drop a few initial path components.

The *git_remote* command works only with git servers supporting the *git archive* command. If the git server does not support it (e.g., *github.com*), it is possible to clone the repository locally and to use the similar *git_local* line. In this case it is possible to specify not only tags or branches but also the specific commits. If the repository is used multiple times to import sources from different directories, it is possible to reuse the same clone. Also, the same clone may be reused, when we have many projects importing sources from the same remote directory.

The VEXTPROJ also offers the *svn* command, allowing to import sources from an SVN directory. If the imported sources are VEXTPROJ compatible, including them into the project is trivial. However, as VEXTPROJ is still a new solution, most repositories do not contain the EPRJ files. In such a case we can create the local EPRJ file, which will add sources imported from the remote repository. Figure 2 shows an example of the project using the OpenCores I2C controller. The VEXTPROJ directly supports only GIT and SVN as remote repositories. However it provides a special line allowing to execute arbitrary command or script[§]:

exec command_or_file_path

That line, supplemented with the appropriate script may be used to download sources from a repository of another kind. Of course, this line is much more powerfull. It may be also used to perform certain preprocessing of sources. The user must be aware that this line may pose a security risk when used with EPRJ file imported from a remote untrusted repository.

[§]The script is executed in the directory, where the EPRJ file containing the appropriate *exec* line is located.

5. IMPORT OF MODIFIED SETTINGS

The project created by VEXTPROJ may be used just to generate the firmware (using the *eprj_build.tcl* script), but it may be also used for interactive work with the design. If the developer introduced any significant changes to the project, it is important that they are stored in the project description. Unfortunately, the Vivado environment does not allow easy separation of changes of settings introduced during the interactive session. It is the user's responsibility to transfer the changed settings into the *proj_def.tcl* file. To help the user, whenever the project is rebuilt from the VEXTPROJ description, the system uses the standard *write_project_tcl* command[¶] to save the initial state od the project to the *initial_state.tcl* file. When finishing the interactive work with the project, the user may execute the *write_project_tcl* command again, storing the current state of the project to another file. Afterward, the file-comparison tools, like *meld* may be used to identify the intentionally changed settings already in the form of the appropriate Tcl commands. Those commands may be then added to the *eprj_create.tcl* script, or may be used to modify the *proj_def.tcl* file.

6. RESULTS

The VEXTPROJ system has been successfully used to describe a few firmware projects associated with the preparation of CBM experiment. Currently, the most complex of them is the protocol[20] tester for STS/MUCH-XYTER2 ASIC. This design uses the STS/MUCH-XYTER2 model developed at AGH that is implemented in System Verilog and stored in a separate GIT repository. Additionally, this model is selected for OOC synthesis so that it is not resynthesized if other parts of the tester are modified. This design also uses a few Xilinx IP cores in the XCIX format.

Because the protocol tester project is not Open Source, to allow verification of the VEXTPROJ itself, two simple projects have been created and published together with the VEXTPROJ scripts on GitHub.[21] The projects have been prepared for a cheap Z-Turn board,[22] but they may be easily ported to other boards based on Xilinx Zynq FPGAs (e.g. Zybo[23] or Zedboard[24]). Those designs implement simple systems using the IPbus or Wishbone[25] bus to connect a small set of registers to the AXI bus controlled by the ARM processor. The designs allow testing of VEXTPROJ with hierarchical projects, with blocks selected for OOC synthesis and also with Block Design sources. Both projects have been successfully compiled and verified.

The development of VEXTPROJ has exposed a few problems in current versions of Vivado environment. In some cases it was possible to implement viable workarounds - e.g., for the problem with incorrect sources compilation order in blocks selected for OOC synthesis.[26] Unfortunately, some of those problems are still unsolved - e.g. incorrect compilation order in the simulator if a certain block is selected for OOC synthesis. Hopefully, next versions of Vivado should be free of those bugs.

7. CONCLUSIONS

The described VEXTPROJ system allows describing the Vivado project consisting of sources imported from different VCS repositories (both local and non-local), to track changes introduced during the development and to rebuild project in a reproducible manner.

The description of the project itself may also be easily controlled by a VCS system. The implementation of VEXTPROJ backend for Vivado is a small set of the Tcl scripts, which can also be controlled via VCS. Therefore the end-user may easily improve the system and adapt it to his or her needs.

VEXTPROJ supports easy reuse of IP blocks, both - delivered in packaged form (XCI or XCIX) and delivered only in source form. It is possible to reuse blocks maintained independently, and distributed via remote repository. The desired version (tag or commit) of sources may be specified.

The VEXTPROJ is an Open Source solution. It is freely available on github[21] under the permissive license. The system is distributed together with two demo projects, which demonstrate its advanced features.

The system seems to work correctly. However, it is a relatively new solution, so more testers are needed. Future improvements should include the development of the backends for other environments or tools (e.g., Altera Quartus and GHDL[27,28]).

[¶]This command is used to create the Tcl script that is able to recreate the project from scratch. Therefore this script contains all settings used by the project in a form of Tcl commands.

ACKNOWLEDGMENTS

The author would like to thank colleagues from CBM collaboration, especially Dr. Walter F.J. Müller from GSI and Dirk Hutter from FIAS for discussions and suggestions related to the ideas presented in this paper.

REFERENCES

[1] "Quartus II Scripting Reference Manual." https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/tclscriptrefmnl.pdf.

[2] "Vivado Design Suite Tcl Command Reference Guide." http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug835-vivado-tcl-commands.pdf.

[3] "Vivado Design Suite User Guide: Using Tcl Scripting." http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug894-vivado-tcl-scripting.pdf.

[4] "Apache Subversion, Enterprise-class centralized version control for the masses." http://subversion.apache.org/.

[5] "git - ditributed even if your workflow isn't." https://git-scm.com/.

[6] "Mercurial - Work easier, Work faster." https://www.mercurial-scm.org/.

[7] "Bazaar." http://bazaar.canonical.com/en/.

[8] "Vivado Design Suite - HLx Editions Productivity. Multiplied.." http://www.xilinx.com/products/design-tools/vivado.html.

[9] "Altera wiki - Version Control." http://www.alterawiki.com/wiki/Version_Control.

[10] "Using Vivado Design Suite with Revision Control." http://www.xilinx.com/video/hardware/vivado-design-suite-revision-control.html.

[11] "Using Vivado Design Suite with Version Control Systems." http://www.xilinx.com/support/documentation/application_notes/xapp1165.pdf.

[12] "Vivado Design Suite User Guide Design Flows Overview." http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug892-vivado-design-flows-overview.pdf#nameddest=UsingSourceControlSystemsWithTheVivadoTool.

[13] "Vivado Design Suite User Guide: Designing with IP." http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug896-vivado-ip.pdf.

[14] "Vivado design suite user guide: Creating and packaging custom ip." http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug1118-vivado-creating-packaging-custom-ip.pdf.

[15] "IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows," *IEEE Std. 1685-2014* (2014). http://dx.doi.org/10.1109/IEEESTD.2014.6898803.

[16] Larrea, C. G., Harder, K., Newbold, D., Sankey, D., Rose, A., Thea, A., and Williams, T., "Ipbus: a flexible ethernet-based control system for xtca hardware," *Journal of Instrumentation* **10**(02), C02019 (2015).

[17] Friman, B., Höhne, C., Knoll, J., Leupold, S., Randrup, J., Rapp, R., and Senger, P., [*The CBM physics book : compressed baryonic matter in laboratory experiments*], vol. 814 of *Lecture Notes in Physics*, Springer, Berlin [u.a.] (2011). also as eBook (ebook ISBN: 978-3-642-13293-3).

[18] Mueller, W. F. J., "PDP-11/70 CPU core and SoC." http://opencores.org/project,w11.

[19] "Using git with PlanAhead." https://forums.xilinx.com/t5/Design-Methodologies-and/Using-git-with-PlanAhead/m-p/270398/highlight/true#M1514.

[20] Kasinski, K., Zabolotny, W., and Szczygiel, R., "Interface and protocol development for STS read-out ASIC in the CBM experiment at FAI R," in [*Proc. SPIE*], Romaniuk, R. S., ed., **9290**, 929028 (Nov. 2014).

[21] "VEXTPROJ - the version control friendly system for creation of Vivado projects." https://github.com/wzab/vextproj.

[22] "Z-turn Board." http://www.myirtech.com/list.asp?id=502.

[23] "Zybo Zynq-7000 ARM/FPGA SoC Trainer Board." http://www.xilinx.com/products/boards-and-kits/1-4azfte.html.

[24] "ZedBoard." http://zedboard.org/product/zedboard.

[25] "SoC Interconnection: Wishbone." http://opencores.org/opencores,wishbone.

[26] "Vivado: incorrect automatic compilation order in OOC synthesis." https://forums.xilinx.com/t5/Synthesis/Vivado-incorrect-automatic-compilation-order-in-OOC-synthesis/m-p/698402.
[27] "GHDL Where VHDL meets gcc." http://ghdl.free.fr/.
[28] "GHDL - a VHDL simulator." http://sourceforge.net/projects/ghdl-updates/.