Version Control Friendly Project Management System for FPGA Designs

Wojciech M. Zabołotny
Institute of Electronic Systems, Warsaw University of Technology

Agenda

- Typical workflow in current FPGA tools
- Need for VCS system
- Incompatibilities between VCS and FPGA tools
- Proposal of solution
- Details of implementation
- Features
- Conclusions and future plans

Typical workflow for FPGA design

- The tools are usually oriented on working in GUI using the “Projects”
- It is very convenient for the developer, who can find most useful options in menus (however usually there are some commands/options that are not available from menus, but only from the Tcl console command line)
- The project stores all the settings in the huge file, usually in the XML format.
- The problem is: How to track changes in the project?

Need for VCS

- Working on a huge FPGA-based system requires VCS!
- When experimenting with project settings, we must be able to isolate the changes which finally made our design working (especially if each recompilation takes 5 hours and we made a few changes in a single iteration)
- When our project contains fragments developed by many teams, we must be able to separate changes introduced by different developers.

FPGA tools and VCS

- Unfortunately most modern FPGA environments are “VCS unfriendly”
  - State of the project is stored in huge, automatically generated files (XML or binary) and tracking of changes is almost impossible
  - Even some sources are stored in the binary files (e.g. the XCIX files used by Xilinx Vivado)
- The vendors themselves see that problem and publish some suggestions:
  - Altera: http://www.alterawiki.com/wiki/Version_Control
  - Xilinx: https://forums.xilinx.com/t5/Design-Entry/Vivado-and-version-control/td-p/347941
- However no perfect solution seems to be available. That justifies an effort to create it...

VEXTPROJ - origin of the name

- In the “old good times” of Xilinx ISE we had PRJ project files. Unfortunately, their meaning was gradually reduced, and Vivado does not use them.

  We try to revive them in an extended version...
- Initially – Vivado EXTended PROJect
- It is likely, that VEXTPROJ will be reused for FPGAs of other vendors (Altera?) or for simulators, so it should be rather understood as

  Versatile EXTended PROJect

VEXTPROJ requirements

- The project was inspired by the needs of firmware development for the CBM experiment.
- We want to have a minimal description of the design, which may be efficiently controlled via VCS (Git, SVN, anything else) and which allows to recreate the Vivado project and rebuild the design.
- We want to be able to keep the sources (HDL, constraints) of the design in different VCS repositories (some of them may be managed independently) and to easily select the version of the sources we currently use to build our design.
- We want to be able to adapt the description to different structures of the repository.
- We want to be able to reuse different IP blocks with maximal flexibility and minimal effort, even if those IP blocks are not packaged as “IP cores”.

What is VEXTPROJ

- It is not a set of Tcl scripts – The scripts are just an implementation. Hopefully in the future it will be implemented in Python (?).
- It is not a format of EPRJ files – this format was changing in the past and is likely to change in the future
- It is supposed to be a methodology.
- Of course this solution is based on ideas borrowed from many people. Just to mention a few:
  - http://xillybus.com/tutorials/vivado-version-control-packaging
  - http://www.fpgadeveloper.com/2014/08/version-control-for-vivado-projects.html
  - http://electronics.stackexchange.com/questions/59477/using-svn-with-xilinx-vivado

Use of IP blocks in VEXTPROJ

- The user of the IP block is not interested in its internal structure.
  - In the standard HDL based approach, the user must manually add all sources to the project.
  - The packaged IP cores solve that problem, but is associated with significant limitations (e.g. no complex types in ports, no parametrized instantiations)
  - In the VEXTPROJ based approach the user may simply include the main EPRJ file describing the block, and instantiate its top entity (complex types are available, parametrization may be used)

How to use the IP blocks with VEXTPROJ

- The typical implementation uses the EPRJ files located in the source tree of the included IP block. To include the block, one needs just to include the main EPRJ file located in the top directory of the sources...
- In case of the independently developed sources which do not support VEXTPROJ, we may create a set of EPRJ files, or even a single EPRJ file describing the sources creating the IP block.
- What is the EPRJ file?
  - The syntax is described on the website

The main EPRJ file

```
include bridge
include ipbus
include constr
include lfsr/lfsr_ooc.eprj
include bd
```

That’s all what’s needed to include the ipbus block

The ipbus/main.eprj file

```
vhdl work ipbus_fabric.vhd
vhdl work ipbus_package.vhd
vhdl work slaves.vhd
include decoder
include slaves
```

The ipbus/decoder/main.eprj file

```
exec addr_dec.py
vhdl work ipbus_addr_decode.vhd
```

The ipbus/slaves/main.eprj file

```
vhdl work ipbus_ctrlreg.vhd
vhdl work ipbus_ctrlreg_v.vhd
vhdl work ipbus_ram.vhd
vhdl work ipbus_reg.vhd
vhdl work ipbus_reg_types.vhd
```

Lines supported by VEXTPROJ

- HDL related
  - vhdl library file_name
  - verilog file_name
  - sys_verilog file_name
  - header file_name
  - global_header file_name
- Other sources
  - bd file_name
  - xci library file_name
  - xcix library file_name
  - mif library file_name
- Constraints
  - xdc file_name
  - xdc_ooc file_name
- Hierarchy related
  - include directory_path
  - include eprj_file_path
  - ooc stub_file_name blk_top_entity
- VCS related
  - git_remote repository_url tag_name [exported_directory [stripped_comp_num]]
  - git_local clone_directory commit_or_tag_name [exported_directory [stripped_comp_num]]
  - svn repository_url_with_exported_path [revision]
- Special command
  - exec program_or_script_path

Splitting designs in separately synthesized parts

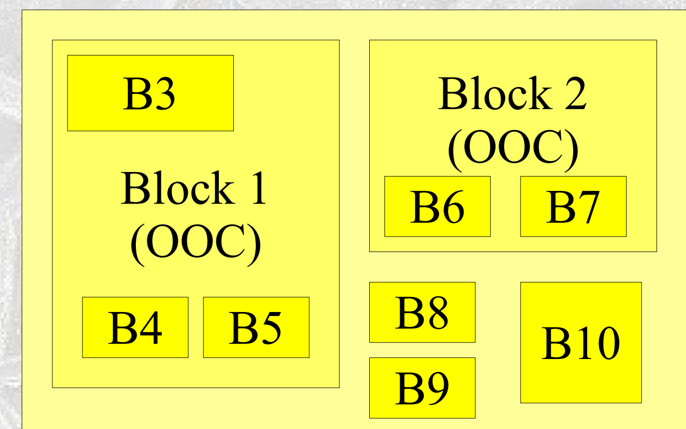
- In a complex design it is important, that it can be split into parts synthesized independently
- In the IP-core based approach, this may be handled by selecting the instantiated component to be compiled in Out-Of-Context mode (of course all the IP-core limitations apply)
- In the VEXTPROJ based approach the same is also possible.

Example of OOC compilation of IP-block with vextproj

- The command to be used:

  ooc [no]auto ip_block_ooc.eprj ip_bl_entity

- This command automatically creates a synthesis run for the IP blocks selected for OOC synthesis
- What about possible problems:
  - Ports with complex types [1]
  - Parametrized instantiation



Using complex types in OOC blocks

- If complex types are used in blocks selected for OOC synthesis, Vivado is not able (yet?) to create “stubs”.
- We must create stubs ourselves and add them to the list of sources

Handling OOC blocks with complex types in VEXTPROJ

"Stub" file

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
library work;
use work.ipbus.all;

entity lfsr_test is

  port (
    clk        : in  std_logic;
    reset      : in  std_logic;
    ipbus_in   : in  ipb_wbus;
    ipbus_out  : out ipb_rbus
    );


end entity lfsr_test;

architecture stub of lfsr_test_a is
attribute syn_black_box : boolean;
--attribute black_box_pad_pin : string;
attribute syn_black_box of stub : architecture is true;
begin
end;
```

EPRJ line defining the OOC block

```
ooc noauto lfsr_test_ooc.eprj lfsr_test
vhdl xil_defaultlib lfsr_test_stub.vhd
```

Important remark!
The stub must be put into the xil_defaultlib library. Otherwise it won't be recognized as the stub, but as a normal implementation of the block.

Handling parametrized OOC blocks

Wrapper for parametrized instance

```
library IEEE;
[...]
entity lfsr_test_a is
  port (
    clk       : in  std_logic;
    reset     : in  std_logic;
    ipbus_in  : in  ipb_wbus;
    ipbus_out : out ipb_rbus
    );
end entity lfsr_test_a;

architecture rtl of lfsr_test_a is
  component lfsr_test is
    generic (
      width : integer range 1 to 32;
      poly  : integer);
    port (
      clk       : in  std_logic;
      reset     : in  std_logic;
      ipbus_in  : in  ipb_wbus;
      ipbus_out : out ipb_rbus);
  end component lfsr_test;
begin
  lfsr_test_1: lfsr_test
    generic map(
      width      => 4,
      poly       => 12
      )
    port map (
      clk       => clk,
      reset     => reset,
      ipbus_in  => ipbus_in,
      ipbus_out => ipbus_out);
end architecture rtl;
```

Parametrized block

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
library work;
use work.ipbus.all;
use work.ipbus_reg_types.all;

entity lfsr_test is
  generic (
    width : integer range 1 to 32 := 11;
    poly  : integer               := 3);
  port (
    clk       : in  std_logic;
    reset     : in  std_logic;
    ipbus_in  : in  ipb_wbus;
    ipbus_out : out ipb_rbus
    );
end entity lfsr_test;
```

EPRJ line defining the OOC blocks

```
ooc noauto lfsr_test_a_ooc.eprj lfsr_test_a
ooc noauto lfsr_test_b_ooc.eprj lfsr_test_b
vhdl xil_defaultlib lfsr_test_a_stub.vhd
vhdl xil_defaultlib lfsr_test_b_stub.vhd
```

Working with VCS

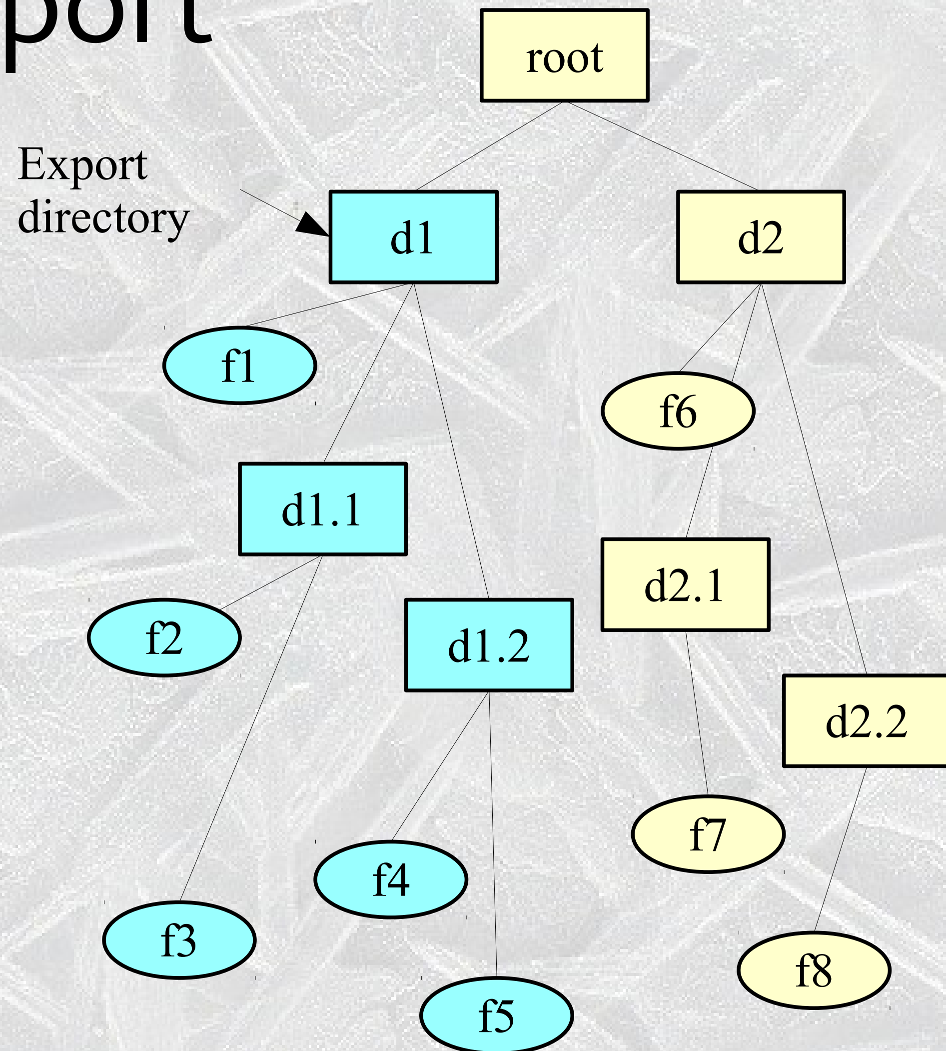
- We assume that our project description, and our sources are controlled via VCS (e.g. they are a working copy of our repository).
- However in a collaboration we must be ready to use sources maintained by other teams and groups, who have their own repositories.
- We may even reuse open source blocks released by people not associated with our project at all.
- Therefore we need a flexible way to import sources from various VCS repositories organized in different ways.
- The idea of implemented solution is roughly based on the implementation of Buildroot environment used to build the embedded Linux from sources imported from multiple repositories

Working with different VCS

- Currently VEXTPROJ supports GIT and SVN:
  - git_remote repository_url tag_name [exported_directory [stripped_comp_num]]
  - git_local clone_directory commit_or_tag_name [exported_directory [stripped_comp_num]]
  - svn repository_url_with_exported_path [revision]
- Support for other VCS may be added by “exec” line, which allows to execute arbitrary program/script in the currently processed directory

GIT support

- We can request particular commit or tag (if not specified, HEAD will be used – which is dangerous)
- It is assumed, that we need only a subtree of the repository, therefore we can specify an “export directory”
- The imported sources will be put in the “ext_src” directory
  - To avoid putting adding them to the project repository, we should create the .gitignore file with single “ext_src/” line.
  - To avoid having too long path names, we may request to drop a few initial path components (e.g., root/d1)



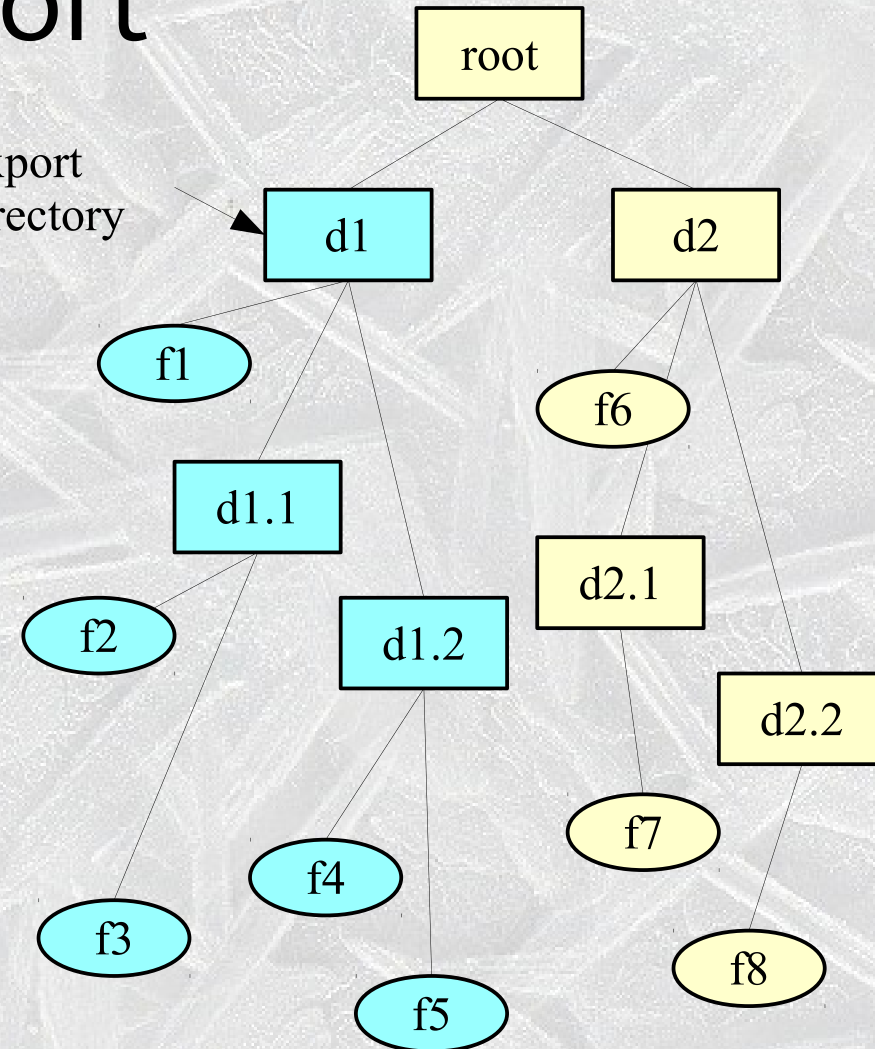
- If the remote git repository does not support “git archive” command, we may prepare the local clone, and export the needed subtree with the git_local line

GIT support

- The above organization of GIT support allows to keep the project description either in the individual repository or as a subtree of a bigger repository.
- Also the independently developed/maintained sources may be kept in the individual repositories or e.g. in a huge repository containing all contributions of the certain team to the particular project or group of projects.
- We avoid unnecessary duplication of the repositories.

  If possible, we may avoid cloning of the repository at all.

  If cloning is necessary, we may clone each “alien” repository only once, and use git_local lines to import necessary sources without generating additional net traffic/external server load.



Example of import from SVN

- The SVN is handled with a single command:
  svn repository_url_with_exported_path [revision]
- We can request particular commit or tag (if not specified, the newest version will be used – which is dangerous)
- The imported sources will be put in the “ext_src” directory
  - To avoid putting adding them to the project repository, we should create the .gitignore file with single “ext_src/” line.
- Example – OpenCores I2C controller (requires having an OC account):

```
svn  http://opencores.org/ocsvn/i2c/i2c/trunk/rtl/vhdl 76

vhdl work ext_src/vhdl/i2c_master_bit_ctrl.vhd

vhdl work ext_src/vhdl/i2c_master_byte_ctrl.vhd

vhdl work ext_src/vhdl/i2c_master_top.vhd
```

Non-trivial use of VCS & OOC support

- Using the VCS and OOC support we may adapt our design even for such non-trivial situation as using two different versions of the same IP block (that should not happen in the final code, but often happens in the development)
- If our module “A” needs the version 1.1 of the module “X” and our module “B” needs the version 1.2 of the module “X”, we may just select modules “A” and “B” for OOC synthesis and import the sources of module “X” independently in the required version into each OOC fileset.
- Without that, the similar problem requires playing with libraries in the source code in VHDL...

How to update design description after the interactive session with GUI?

- The big advantage of Vivado GUI is the possibility to interactively play with settings.
- Changes in sources and constraints should be handled automatically (in case if new files are created, remember to move them to sources directory and add them to VCS!)
- Unfortunately the changed settings are stored in the XML "project_name.xpr" file, and changes are difficult to extract.
- The suggested approach is to:
  - write the project to the Tcl file at the beginning of the session (VEXTPROJ now writes it to the initial_state.tcl)
  - At the end of the session write the project to the Tcl file
  - Review the changes (with "diff" or "meld") and add commands changing the settings to the scripts.
- This is the least mature part of the VEXTPROJ. Any suggestions/contributions are welcome!

Conclusions and future plans

- The VEXTPROJ is available as Open Source (in fact CC0/Public Domain) solution on https://github.com/wzab/vextproj
- The system allows to describe the Vivado project consisting of sources imported from different repositories (both local and non-local), to track changes introduced during the development and to rebuild project in a reproducible manner.
- There are two demo projects provided, that test different advanced features of the system.
- The system seems to work correctly, however more testers are needed...
- *A few features planned for the nearest future are:*
  - *The protection against an attempt to nest the OOC selected blocks (and automatic handling of the OOC flag in the included XCI/XCIX IP cores).*
  - *Possibility to add arbitrary properties to files*
  - *Support for simulations (simulation-only fileset)*
- It is difficult but necessary to a find balance between good support for a particular toolchain (Vivado) and portability to other environments (e.g., Quartus)

Thank you for your attention!