A collaborative HDL management tool for ATLAS L1Calo upgrades

Francesco Gonnella[*]
University of Birmingham
E-mail: francesco.gonnella@cern.ch

Coordinating firmware development among many international collaborators is becoming a very widespread problem in particle physics. Guaranteeing firmware synthesis with place and route (P&R) reproducibility and assuring traceability of binary files is of paramount importance. Our HDL managing tool, developed in Python and tightly integrated with CERN Gitlab and Xilinx Vivado, tackles these issues by exploiting advanced Git features and by paying particular attention to Intellectual Property handling. In LHC Run-3, the ATLAS L1Calo Trigger system will be upgraded with new feature extraction and readout modules and our tool is used for the firmware development for these modules.

Topical Workshop on Electronics for Particle Physics
17 - 21 September 2018
Antwerp, Belgium

[*]Speaker.

1. Introduction

The aim of this project is to fruitfully combine the work of many developers writing different parts of the same firmware, as this is becoming a very widespread problem in high-energy physics. FPGA designs tend to become bigger and there is the need to handle different projects sharing a large amount of code, as for example multiple chips on the same board using the same modules such as SPI, I2C interface or chips having almost the same firmware apart from minor modifications. Moreover, there is often the need to handle third-party code used by many projects but very rarely modified by the developers. An example is the IPbus [2] library, handling communication among FPGAs. We want to fully exploit Git [3] for collaborative development, as it is a very powerful and comprehensive tool and it is the standard chosen by CERN.

Our tool allows developers to use Xilinx Vivado [4] normally, as much as reasonably achievable, allowing the exploitation of Git without adding overhead work. Our main goals are: guarantee firmware synthesis with P&R reproducibility and assure traceability of binary files. These issues are tackled by keeping an absolute control of source files, constraints, and synthesis and implementation settings and embedding the version number and the Git SHA[1] into the firmware binary file. The tool is well integrated with Vivado, with particular attention to the handling of the library of intellectual properties[2] (IP).

Our tool currently handles a subset of boards for L1Calo upgrades: the eFEX, the ROD, and the FTM. Every board contains 1, 2 or 4 FPGAs. About 10 developers are currently working at these projects, sharing many firmware modules. All the projects use the IPbus library for intercommunication and Ethernet connection.

2. HDL on Git

Hog (HDL on Git) is a set of Tcl scripts plus a suitable methodology to exploit Git as an HDL repository. Its main ingredient is a Tcl library (hog.tcl) containing functions for handling file i/o, Git, and integrating into Vivado workflow. Hog also contains a series of shell scripts to help developers to handle common tasks such as initialising the repository[3], creating Vivado projects and handling Xilinx IPs.

Hog scripts are integrated into the Vivado flow and produce infos, messages, warnings and errors that are caught by Vivado. For example a warning appears if the repository is not clean at the moment of synthesis. Hog scripts can also properly handle Mentor Graphics Questa [5] simulations.

The pre-synthesis and the post-bitstream scripts are integrated into Vivado and run automatically at every workflow. The first calculates the repository status and feeds data to the top VHDL files. The second copies all the relevant files created by Vivado during the workflow into specific directories. It renames the generated binary file to include the version of the repository, to ease

[1]A checksum identifier that Git uses to unambiguously identify a commit in a repository.

[2]Xilinx IPs are black box firmware modules, designed by Xilinx. They can be RAMs, FIFOs, Ethernet controllers, transceiver controllers, etc.

[3]After checking out the repository you need to create symbolic links to Hog Git-hooks, handle files that must be ignored by Git, compile simulation libraries

traceability. This is done so as not to rely on developers renaming files manually. Being written in Tcl, in principle Hog can be extended to be compatible with other HDL design suites like Intel Quartus [6].

The repository methodology that Hog users must adopt is to make use of temporary feature branches. These branches are then merged to the master by one or more librarians, always by human intervention. An important thing to notice is that no new-commit must be created when a branch is merged into the master branch. This is fundamental to preserve the Git commit identifier after the merge is done, and it is necessary both for reproducibility and for traceability.

There are special folders for firmware constraints, list files, XML files[4], the firmware top module file, and for Xilinx IPs. Developers can use Vivado in project-mode normally, but they must not add new files to the project via the Vivado GUI, but instead add the file names to the list files and re-create the project. This ensures that the modification is propagated to the repository.

No Vivado-specific file is committed to the repository, as recommended by Xilinx [1]. Instead the project is created locally by a Hog Tcl script. Developers must only maintain and commit one simple Tcl file for each project in the repository. This file contains all Vivado properties such as the FPGA model and the strategies to be used for synthesis and implementation and so on.

The developers must also maintain the list files, that are stored in specific locations in the repository. These are simple text files containing all the source file names to be added to the project together with properties such as VHDL 2008 compatibility or top entity name for the simulation. The files listed in each list file are imported by the create-project script into a different Vivado library. This feature can be used to keep the different functionalities of the firmware well separated.

The create-project script creates a new Vivado project adding to it the files read out from the list files and creates the default runs with properties and strategies described in the project Tcl file. This methodology ensures being Vivado-version independent[5]. As we have said, in order to change settings or add new files, developers must avoid using Vivado GUI and instead modify the Tcl script or list files and recreate the project with Hog script. It is always possible that a developer forgets to add a file to the list files, or a property to the Tcl file. These potential errors are caught by the automatic workflow engine described in Sec. 3.

Xilinx IPs are handled by Hog with particular care. Developers must create out-of-context IPs and store files into a specific directory, whose subdirectories are ignored. This is to avoid committing to the repository all the files that Vivado generates at synthesis or simulation time. In fact, only 2 files are to be committed to the repository: the XCI file, and the XML file (as recommended by Xilinx in [1]). These are text files that can be properly handled by Git. Hog methodology tells the developer how to properly set the `.gitignore` file and to place the IPs in specific location in the repository.

Vivado synthesis is launched over all the IPs in the project, before starting the main synthesis. However when synthesising an IP, Vivado modifies the XML files by changing some metadata (date, time of last synthesis). As a consequence the XML files appear to be modified in the repository and have to be committed or reset to the original state.

[4]XML files are used only in case the repository makes use of the IPbus library. They are IPbus specific and contain information about the firmware address mapping.

[5]Hog was tested on the following Vivado versions: 2016.4, 2017.1, 2017.3, 2018.1, 2018.2.

Hog handles this situation by means of a specially designed pre-commit Git-hook. For every XCI file, the relative XML file is *marked as unchanged* unless the XCI is also modified, ignoring all irrelevant modifications to the XML. A Hog script can also be used to manually *mark as unchanged* all the IP XML files or undo this action, useful when adding or removing IPs.

Hog embeds the repository Git-SHA into firmware registers. Telling which SHA is more recent though, is impossible without looking at the repository. For this reason a numeric version of the form $M.m.p$ (Major, minor, patch) is evaluated and fed to the firmware.

To easily compare different firmware versions, Hog calculates the version of several parts of the project and embeds them into the firmware. For the submodules, only the Git-SHA can be evaluated. The values of M, m and p are automatically extracted from Git tags. The tags are automatically created when official versions are released, so as not to rely on developers to increase numbers manually in VHDL files.

For every project, Hog evaluates the Git-SHA and possibly the version independently for various parts of the repository such as: the Top directory (top VHDL file, constraints, list files, project Tcl script), IPbus XML files, submodules (for every submodule in the project the Git-SHA is included), source libraries (one SHA and one numeric version for each library, corresponding to a list file. Finally the global version of the repository is evaluated and also used as official version identifier. The time and date of the last commit are also embedded in firmware registers. We do not use the synthesis time as it differs from one synthesis to the next, preventing reproducibility.

Hog is well integrated with the IPbus system: it handles IPbus XML files at synthesis time by adding the XML version and Git-SHA as a special flag to the version XML file. This allows IPbus software to verify if the version of the XML file in use is compatible with the firmware in the device. Moreover, in the post-synthesis script, Hog collects all project XML files into the same directory, and prepares them to be distributed together with the binary file.

3. Automatic Workflow Engine

The automatic workflow engine, referred to as awe, is a Python program running on CERN OpenStack virtual machines. It is able to launch Vivado in text-only mode to run a complete firmware workflow and produce binary files. Awe is connected to CERN Gitlab through web-hooks and it is triggered by Git merge-request (MR) events such as: opening a non-work-in-progress MR to the master branch or pushing on a branch for which a MR is already open.

When triggered, awe updates and cleans the repository. As a second step, it merges the master branch onto the branch for which the MR was opened: this is essential to preserve the Git-SHA when the merge request is accepted, as explained in the following. If the automatic merge fails, human intervention is needed, and the developers are notified through Gitlab notes.

Awe then compares the master branch with the feature branch and finds out which projects were changed and prepares to launch Vivado for those projects only. Then awe launches the Vivado flow on the modified projects.

When the design flow is over and successful, awe writes a MR note on Gitlab (using Gitlab API) containing a summary of Vivado timing and utilization reports and runs Doxygen to produce up-to-date firmware documentation. Finally, awe approves the MR allowing the librarians to merge it, possibly after further tests. When a MR is merged by the human librarian, awe copies the properly renamed binary files, the IPbus XMLs, and the Doxygen documentation into the official website connected via EOS to the virtual machine.

Awe is also integrated with IPbus. Before starting a Vivado workflow, it compares IPbus XML files with the VHDL address decode files and reports the difference output as a Gitlab note.

Hog extracts the values of M, m, and p from Git tags and feeds these to the firmware registers via VHDL generics at synthesis time. But, in order to do this, it must know the new firmware version a priori, before starting the synthesis, and before accepting the merge request. How to do this? The key is that before starting the firmware workflow, the master branch must be merged onto the feature branch. This creates the commit that will be preserved even when the MR will be accepted and pushed to master. In practice, awe extracts the current version number from the most recent tag describing the current commit and increases it, creating a beta tag of the form bx-v$M.m.p$-n. Where x is the Git merge request number and n is the number of pushes made to a branch after the opening of the merge request. Modifications can happen after the MR because the automatic workflow has failed or simply because the developer decides to make some changes after the automatic workflow is completed. Official tags, instead have the simple form of v$M.m.p$.

4. Conclusions

The Hog tool is available on `http://gitlab.cern.ch/atlas-l1calo-efex/Hog`, it is well integrated with Vivado with minimal overhead work for developers. It guarantees synthesis and P&R reproducibility and binary file traceability by embedding Git-SHA and a numerical version into the firmware. It also handles properly Xilinx IPs committing only relevant files.

Awe is currently in operation on CERN virtual machines (32 CPUs, 64 GB RAM, 1 TB storage) for some atlas L1Calo upgrade boards: the eFEX, ROD and FTM. The awe software is available on `gitlab.cern.ch/atlas-l1calo-efex/awe`.

Both Hog and awe are integrated with the IPbus system. The eFEX website `cern.ch/efex` and eFEX firmware repository `gitlab.cern.ch/atlas-l1calo-efex/eFEXFirmware` are working examples of how these tools operates.

References

[1] Xilinx, XAPP1165 (v1.0) August 5, 2013

[2] C. Ghabrous Larrea et al., *IPbus: a flexible Ethernet-based control system for xTCA hardware*, JINST **10** (2015), C02019

[3] `git-scm.com/`

[4] `www.xilinx.com/products/design-tools/vivado.html`

[5] `www.mentor.com/products/fv/questa/`

[6] `www.intel.com/content/www/us/en/software/programmable/quartus-prime/`