A collaborative HDL management tool for ATLAS L1Calo upgrades

Francesco Gonnella - University of Birmingham

TWEPP 2018 – Antwerp, Belgium
Fruitfully combine the work of many developers writing different parts of the same firmware

Handle different projects sharing a big amount of code
  - e.g. multiple FPGAs on the same board

Handle third-party code used by many projects but never (or rarely) modified by the developers
  - e.g. IPbus: http://cern.ch/ipbus

Fully exploit Git for collaborative developing
  - Git is the standard chosen by CERN: http://gitlab.cern.ch
Address the problem of coordinating collaborative firmware development

- Allow developers to use **Vivado normally** (as reasonably achievable)
- Allow a fruitful usage of Git
- Do not add overhead work

Guarantee firmware synthesis with P&R **reproducibility**

- Absolute control of HDL files, constraint files, Vivado settings

Assure **traceability** of binary files:

- **Embed Git SHA** into firmware registers
- **Version number** must be evaluated automatically and **embedded in firmware** registers

Exploit **Git** features and integrate with **Vivado** with **IP handling**

- Cannot use Gitlab CI as it is not suitable for firmware workflow

Currently used by L1Calo eFEX, FTM, ROD in ATLAS upgrades
Our case: a subset of boards for L1Calo upgrades

- ~10 developers
- 3 different boards: eFEX, ROD, FTM
  - 1, 2 or 4 FPGAs
  - Sharing significant amount of code
- All using IPbus system (our tool is specifically designed to handle it)
Designed systems: **Hog and awe**

In order to tackle these issues 2 systems have been designed:

1. **HDL on Git (Hog)**: a set of scripts and methodologies to safely store HDL firmware files on Git
   
   Developers add the scripts to the repository (as a submodule) and respect a set of rules.

2. **Automatic Workflow Engine (awe)**: a python program to automatically synthesise and implement HDL projects when a Git Merge Request is opened

   Runs on CERN Virtual Machine and interacts with Gitlab.

   - 3 repositories: **eFEX, FTM, ROD** all using: **Hog** scripts and methodologies and including **IPbus** as a submodule.

   - 3 CERN VMs are running **awe** for these repositories
What is Hog?

- **Hog** (HDL on Git) is a set of **Tcl/Bash** scripts plus a suitable methodology to exploit Git as an **HDL repository** and guarantee:
  - Synthesis **reproducibility**
  - Binary file **traceability**

- It contains a Tcl library (hog.tcl) and series of **Tcl and shell scripts** to handle HDL code in the repository and interface it with Vivado

- **Command line scripts**
  - Initialize repository (link Git-hooks, handle ignored files, compile libraries, generate projects)
  - Create project script (Generate projects from list files)
  - IP handling scripts (see next slides)

- Scripts **integrated in Vivado** flow and Questasim
  - Pre synthesis script (feed versions, Git SHA, date and time into HDL generics)
  - Post write bitstream script (copy relevant file in specific directories)

- **Hog** is a **simple project**:
  - **Minimal overhead** work to developers
  - Relies on file/directory names and structure
  - Being written in Tcl, Hog can be extended to be compatible with other HDL design suites like Intel Quartus
Use of temporary feature branches:
- To add a feature/debug, every developer creates a new temporary branch starting from the master branch.
- Branches are then merged to master by one or more librarians.

No new-commit on merge to master (important to preserve commit SHA).

Include scripts (Hog) directory in your repository:
- As a submodule, to keep it up to date.

Create Vivado projects using the `CreateProject` script:
- Project files will be created in a directory outside of the Git repository.

Choose file names and keep them in folders using Hog prescriptions:
- Special folder for: constraints, list files, xmls, project top VHDL file.
- Special folder for Xilinx IP.

The developers use Vivado in project-mode normally, with few exceptions:
- Do not add new files to the project, but add file names to the list files and re-create the project.
- Create out-of-context IPs and store files (xml and xci) into a specific folder.
Directory tree (2/2)

- TOP
  - <project 1>
    - Xdc
      - Constr1.xdc
      - Constr2.xdc
    - xml
      - xml.lst
    - List
      - Algolib.src
      - Infra.src
      - Readout.src
      - Algo.sim
      - ipbus.sub
      - XDC.con
  - <Project 2>
  - <Project 3>

Constraint files
IPbus xml list file
Project top VHDL file
Project Tcl file
Library list files
Simulation list file
Submodule list file
Constraint list file
Creating Vivado project

- No Vivado-specific file is committed to the repository, the project is created locally by a Hog Tcl script

- Maintain and commit **one simple Tcl file per project and list files**
  - Creates the projects adding the files listed in the list files
  - Creates default runs with **properties** and **strategies**
    - Including report strategies since 2017.3

- Vivado **version-independent** (from 2016.4 onwards)

- To change settings, **modify the Tcl script** and **recreate the project**
  - Potential errors caught by **awe**

- **List files** (i.e. txt files containing source file names) are read by Hog Tcl script
  - When adding a new file to the project, **modify list file, recreate the project**
    - If files are added directly to the project, the modification cannot be propagated to the repository
    - Potential errors caught by **awe**
Projects contain **Xilinx Intellectual Properties** (IP)
- FIFO, RAM, MGT, etc.

Each made of **multiple files** (VHDL, Verilog) contained in a directory

Only the 2 main files are committed to the repository: `xci`, `xml`
- These are text file that can be handled by Git

**All the other files are generated** by Vivado at synthesis/simulation time and must be ignored 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 in multi thread over all the IPs in the project, before starting the main synthesis.

When synthesizing an IP, Vivado modifies the xml files by changing metadata (date, time of last synthesis).

There are two unpleasant consequences of this:
1. The repository appears to be modified (dirty) before every synthesis.
2. The xml files appear to be modified in the repository and have to be committed or reset to the original state.

Hog handles this situation with a 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.
In order to guarantee bitfile traceability, Hog embeds the repository Git-SHA/Version into firmware registers.

To easily compare different firmware versions we need to calculate the version of each firmware library and submodule and embed them.

Git can evaluate SHAs independently for any given subset of files:
- Given 2 SHAs, it is impossible to tell which is more recent without looking at the repository.
- For this reason a numeric version M.m.p is evaluated by awe to indicate how recent each library is, as explained in the following slides.

M, m and p are automatically extracted from Git tags and fed to firmware via VHDL generics at synthesis time.

Releases tags are of the form: v<M>.<m>.<p>

Official versions are automatically tagged by awe, not to rely on developers to increase numbers manually in VHDL files.
For every project evaluate Git SHA and version independently for the following parts:

- **Top:** all that is contained in the project top directory: top vhdl file, constraints, list files, project tcl script (containing synthesis and implementation settings)
- **IPbus xml:** version of the IPbus address map XML
- **Submodules:** for every submodule in the project the Git-SHA is included
  - E.g. eFEX, FTM have IPbus as a submodule
- **Libraries:** one SHA and one M.m.p version included for each library (corresponding to a list file) in the project
  - E.g. eFEX processor FPGA contains: Algorithm, Infrastructure, Readout
- **Global:** the global version of the repository, used as official version identifier

Firmware compilation time and date are embedded in firmware registers

An additional register called OFFICIAL contains info on the status of the repository at the moment of synthesis

- Note that synthesis can be done locally by developers so it can be unofficial, repository might not be clean and so on
An example of versions

- Here is the pre-synthesis script output for the eFEX board, processor FPGA project
- Date, time, SHAs and versions are evaluated
- Version register are formatted in hex as MM mm pppp

------------------------- PRE SYNTHESIS -------------------------
03/03/2018 at 00:23:35
Firmware date and time: 03 03 2018, 00 00 23 35
Global      SHA: D04AC65, VER: 00 01 0022
xml         SHA: 47EAD9D, VER: 00 01 0022
Top         SHA: 47EAD9D, VER: 00 01 0022
Official reg: C0000001
--- Submodules ---
IPBus       SHA: 27a4775

--- Libraries ---
TOB_rdout_lib SHA: 75D668F, VER: 00 01 0001
algolib      SHA: 86E8CDF, VER: 00 01 0022
infrastructure_lib SHA: 86E8CDF, VER: 00 01 0022

-----------------------------------------------------------------
What is awe?

- **awe** (automatic workflow engine) is a **Python** programme running on CERN **Openstack** Virtual Machines

- Launches Vivado in text only mode (with CERN license) to produce bitfiles

- Connected to **gitlab.cern.ch** through web-hooks
  - Triggered by **merge request events**
  - Opening a non WIP (work in progress) merge request to master
  - Pushing on a branch for which a MR is open
  - Runs **complete design-flow** and produce bit/bin files, reports, and xmls
  - **Automatically extracts Version from Git tag** `vM.m.p` and increases it:
    - `m` if `merge-request` title starts with “MINOR_VERSION:”
    - `M` if `merge-request` title starts with “MAJOR_VERSION:”
    - `p` in all the other cases

- **Write notes on Gitlab** with Vivado reports using Gitlab API

- Automatically tags commit if design-flow is successful
  - Add **timing** and **utilisation** report info in the TAG message

- Runs Doxygen producing up-to-date documentation
Awe flow in detail

- Wait for merge request (MR) event

- If MR satisfies criteria:
  - **Update** and clean the repository
  - **Merge master** branch (if automatic merges fails, human intervention is needed)
  - **Compare** and find out which **projects** to implement
  - Evaluate **version** from most recent tag
  - **Tag new beta TAG** and push (needed to preserve SHA):
    - If v1.2.3 make b42v1.2.4-1 (42 is the mr number, 1 is the first attempt)
    - If b42v1.2.4-5 make b42v1.2.4-6 (just increase the attempt number)
  - For each project, run Vivado and produce bit files
  - Run **Doxygen**
  - **Approve merge** request, enabling the librarian to merge it

- When a merge request is merged (by human intervention)
  - Copy the **Doxygen documentation** in the official website
  - Tag the **official version** (e.g. v1.2.4)
  - **Write a release note** with work-flow results

Francesco Gonnella - University of Birmingham | 20 September 2018
Awe extracts the values of $M, m, p$ from Git tags and feeds these to the firmware registers via VHDL generics at synthesis time.

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?**

- Awe extracts the **current version number from the most recent tag** describing the current commit and increases it, creating a **beta tag**
  1. Official tags: $vM.m.p$
  2. Beta tags (candidate for version): $bx-vM.m.p-n$
     - $x$ is the Git merge request number (to avoid duplicated tags)
     - $n$ is the number of attempts made after the opening of the merge request for that specific version

**Does it have to be so complicated?**
In the simplest scenario a candidate for version 1.2.5 gets merged and becomes Version 1.2.5

Purple commits are created automatically by *awe* and a workflow is started to produce firmware bit files

- Note that the version is known *before* the synthesis starts, in order to be embedded into firmware registers
- Merging master onto the feature branch is necessary to create unambiguous commits SHA
- If the workflow is successful the same commit can be pushed onto master *without creating a new commit*, hence preserving SHA and version number
- In case the automatic merge fails, the developer is required to merge master onto his branch on his own
- To preserve Git SHA, **no new commit** is made upon a merge to master.

- The official version number goes to the branch that gets merged first.
  - In that case another merge of master onto feature branch is needed.
  - The version number is then increased.

- The MR number in the TAG is needed to **avoid duplicated TAGs**.
Hog is available on gitlab.cern.ch/atlas-l1calo-efex/Hog
- Integrated with Vivado with minimal overhead work for developers
- Guarantee workflow reproducibility and bitfile traceability by embedding Git SHA and numerical version into the firmware
- Properly handle Xilinx IPs committing only xml and xci files

awe triggered by Gitlab merge requests, runs complete firmware workflow
- Currently in operation on CERN VM (32 CPUs, 64 GB RAM, 1 TB storage)
- Here gitlab.cern.ch/atlas-l1calo-efex/awe you can find awe python program, setup script for VM, website files

Both Hog and awe are integrated with the IPbus system

Visit the eFEX website: cern.ch/efex and eFEX firmware repository gitlab.cern.ch/atlas-l1calo-efex/eFEXFirmware as a working example
Thanks for your attention

Francesco Gonnella
After the blue merge request is accepted version 1.2.5 is created

In this situation the merging of master branch to the green branch must be redone because the green branch needs to be merged with the new master containing the blue branch

- The bitfiles produced with the green branch for version 1.2.5 have the wrong version embedded, but the workflow needs to be redone anyway
- That commit will never be promoted to version v1.2.5, preventing confusion
Anothermergecase

- Any new commit in master branch is made with a merge from feature branches

- If we **make other commits after the merge request**, we create new attempts for the same version candidate
  - This may be done because the automatic workflow was not successful or just because we had previously forgotten to add something

Francesco Gonnella - University of Birmingham
Gitlab CI are very well suited for software development:

- Building is done with very common tools (e.g. gcc)
- Building/Testing is fast wrt firmware
- Non need for complex interaction with repository

Specific features are not compatible with firmware development:

- Triggered by **push events** on branches rather than merge requests
  - Most of the times useless effort, synthesize a firmware may take hours
- Not enough customizable: **we need to know git SHA and version before starting the synthesis**
- Cannot **push to repository**
- Cannot **interact with merge-request parameters**
  - Cannot check if MR is Work In Progress (WIP)
  - Cannot parse description to get directives
**Hog, awe and IPbus**

- **Hog** is integrated with the **IPbus** system ([http://cern.ch/ipbus](http://cern.ch/ipbus))

- **Hog** handles **IPbus xml files** at synthesis time:
  - Add 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
  - Collect all **project xml files** in the same directory

- **Awe** compares xml files with VHDL address decode files and reports the diff output as a Gitlab Note
While the workflow is running, one can monitor the status on eFEX website: cern.ch/efex
**Documentation with Doxygen**

**VHDL highlighted syntax**

```
Out_TOB_sync : process (CLK280)
begin  -- process dd
if rising_edge(CLK280) then
  if CG_Load = '1' then -- sync IN_load with ClkOut
    SyncLoad <= '1';
  else
    SyncLoad <= '0';
  end if;
  if SyncLoad = '1' then -- Reset the counter accordingly
    SyncCount <= (others => '0');
  else
    SyncCount <= std_logic_vector(unsigned(SyncCount)+1);
  end if;
  if unsigned(SyncCount) = PHASE280 then -- Produce a signal to capture data
    OutLoad <= '1';
  else
    OutLoad <= '0';
  end if;
end if;
end process Out_TOB_sync;
```

---

**HTML documentation**

**Use Clauses**

```
STD_LOGIC_1164
NUMERIC_STD
ipbus_decode_address_table_ALGO
```

**DataTypes**

```
Package <ipbus_decode_address_table_ALGO>
Use ipbus address decode automatically generated from XML.
```

**AlgoDataTypes**

```
Use external algorithm data types and functions.
```

**ipbus_reg_types**

```
Use ipbus library.
```

**Generics**

```
FWHASH     std_logic_vector (31 downto 0) := x"00000000"
Contains adi贞lib ght SHA.
```

**Ports**

```
CLK200    in std_logic
200 MHz clock
CLK280    in std_logic
280 MHz clock, used in the output stage
RESET     in std_logic
Synchronous reset, active high. To be removed in next versions.
IN_Load    in std_logic
40 MHz clock, 20% duty cycle, -36 deg phase
ipb_clk    in std_logic
IPBus clock
ipb_rst    in std_logic
IPBus reset.
ipb_in    in ipb_webus
IPBus write bus.
ipb_out    out ipb_rbus
IPBus read bus.
IN_Data    in AlgolInput
Algorithm external data structure, defined in AlgoDataTypes.vhd.
```

---

Francesco Gonnella - University of Birmingham  
20 September 2018