
Hog

Release 2020.1

F. Gonnella, D. Cieri

Sep 21, 2020

GETTING STARTED

1 Introduction

1

INTRODUCTION

Coordinating firmware development among many international collaborators is becoming a very widespread problem. Guaranteeing firmware synthesis with Place and Route **reproducibility** and assuring **traceability** of binary files is paramount.

Hog tackles these issues by exploiting advanced git features and integrating itself with HDL IDE (Vivado or Quartus). The integration of these tools intends to reduce as much as possible useless overhead work for the developers.

1.1 What is Hog

Hog is a set of Tcl/Shell scripts plus a suitable methodology to handle HDL designs in a Gitlab repository.

Hog is included as a submodule in the HDL repository (a `Hog` directory is always present in Hog-handled repository) and allows developers to create the Vivado/Quartus project(s) locally and synthesise and implement it or start working on it.

The main features of Hog are :

- a fast way to maintain your HDL code on git
- to have automatic tagging for easy versioning
- a template to run a continuous integration in your Gitlab repository
- the possibility of creating multiple project sharing one top level
- to store the output binary files on EOS
- to work both with Windows and Linux

Everything is as transparent as we could think of. Hog is designed to use just a small fraction of your time to set up your local machine and get you to work to the HDL design as soon as possible.

1.2 Rationale

For synthesis and Place and Route (P&R) **reproducibility**, we need absolute control of:

- HDL source files
- Constraint files
- Vivado/Quartus settings (such as synthesis and implementation strategies)

For **traceability**, every time we produce a binary firmware file, we must:

- Know exactly how the binary files were produced

- Be able to go back to that point in the repository

To do this, Hog **automatically** embeds the git **commit SHA** into the binary file together with a more understandable **numeric version M.m.p**. Moreover, it automatically renames the file, including the version and inserts the hexadecimal value of the SHA so that it can be retrieved (using a text editor) in case the file gets renamed.

Avoiding errors is impossible, but the goal of Hog is to leave as little room as possible.

Another important principle in Hog is to **reduce to the minimum** the time needed for an external developer to **get up to speed** to work on an HDL project. For this reason, Hog **does not rely on any external tool** or library. Only on those you must have to synthesise, implement (Vivado/Quartus) and simulate (Modelsim/Questasim) the design.

To start working on any project¹ contained in a Gitlab repository handled with Hog, you just need to:

```
git clone --recursive <HDL repository>
cd <HDL repository>
./Hog/CreateProject <project_name>
```

The project will appear in `./VivadoProject/` (or `./QuartusProject/`) and you can open it with your Vivado (Quartus) GUI.

1.3 What is in the Hog folder

The Hog folder contains plenty of Tcl and Shell scripts.

E.g. you can run:

```
./Hog/Init.sh
```

to initialise the repository locally, following the instructions.

You can always have a look yourself. Most of the scripts have a `-h` option to give you detailed instructions. The most important script is `Hog/CreateProject.sh` that serves to create the Vivado/Quartus project locally. When creating the project, Hog integrates a set of Tcl scripts in the IDE software to handle and guarantee reproducibility and traceability.

1.4 Other folders that you need

A folder called `Top` shall be in the root of the repository and it contains a sub-folder for each Vivado/Quartus project in the repository. Each of these directories has a fixed -easy to understand- structure and contains everything that is needed to re-create the Vivado/Quartus project locally, apart from the source files² that the developer can place in the repository at his/her own convenience.

An IP (and possibly a BD) folder shall be used to store Intellectual Properties (and Board Design). Apart from these few mandatory directories, any structure of subdirectories can be created in the IP (and BD) folder.

¹ If you don't know the project name, just run `./Hog/CreateProject` and a list will be displayed.

² Source files are the HDL files (`.vhd`, `.v`) but also the constraint files (`.xdc`, `.sdc`, `.qsf`, `.tcl`, ...). The IP files (`.xci`, `.ip`, `.qip`, ...) and the Board Design files must be stored in special folders, as explained later.

1.5 Hog user manual

In this website you can find a quick guide to learn how to work in a *Hog-handled repository* or to *set up a new one*, as well as a complete user manual to understand all the details and learn how to maintain a Hog-handled repository.

1.6 Quartus support

In this manual we will refer to Quartus even though **Quartus is currently not fully supported**. We are planning to give full Quartus support in a dedicated release in the future.

1.7 Contacts

Would you like to have fun with git and a Tcl? Please join us and read the *Developing for Hog* section.

To report an issue use the git issues in the [Hog git repository](#). Please check in existing and solved issues before submitting a new issue.

For questions related to the Hog package, please get in touch with [Hog support](#).

For anything related to this site, please get in touch with [Nicolò Biesuz](#) or [Davide Cieri](#)

1.7.1 How to work with an existing Hog-handled repository

This section is intended for a developer that starts working with an existing HDL project that is managed with Hog.

All the instructions below can be executed both on a Linux shell, or on git bash¹ on a Windows machine.

For all of the following to work, Vivado (or Quartus) executable must be in your PATH environment variable: i.e. if you type `vivado` the program must run. If you intend to use Modelsim or Questasim, also the `vsim` executable must be in the PATH: i.e. if you type `vsim` the simulator should start.

Requirements

This is a list of the requirements:

- Have git (version 2.7.2 or greater) installed and know git basics
- Have Vivado or Quartus installed and in the PATH
- Optionally have Questasim installed and vsim in the PATH

¹ To open a git bash session navigate to the directory where you want to open the bash. Right click on the folder and select open git bash here.

Cloning the repository

First of all, you have to clone the repository², let's call it *repo* from now on. Go to the website of the repository, choose the protocol (ssh, git, https, krb5) and copy the clone the link.

```
git clone --recursive <protocol>://gitlab.cern.ch/repo.git
```

Now you have all the source code and scripts you need in the *repo* folder.

Create Vivado/Quartus projects

To start working you can now create the Vivado (or Quartus) project you are interested in, say it's called *project1*. To do that, go into the repository (`cd repo`) and type:

```
./Hog/CreateProject.sh project1
```

This will start a Hog script that creates the Vivado (or Quartus) project in the directory `VivadoProjects/project1` (or `QuartusProjects/project1`). Inside this directory you will find the Vivado `.xpr` file (or the Quartus `.qpf` file).

You can now open the project file with Vivado/Quartus GUI and synthesise/implement the project.

If you don't know the project name, just run `./Hog/CreateProject.sh` and you will get a list of the existing projects present on the repository³.

To create all the projects in the repository, you can run the Hog initialisation script, like this:

```
./Hog/Init.sh
```

This script will also, if you wish, compile Modelsim/Questasim libraries.

Adding a new file to the project

When you open *project1* with Vivado or Quartus, you can **work (almost) normally** with the GUI.

The CreateProject script, that you have just run, has integrated Hog's Tcl scripts in the Vivado/Quartus project. From now on, Hog scripts will run automatically, every time you start the synthesis or any other step in the workflow. In particular, the pre-synthesis script will interact with your local git repository and integrate its version and git commit SHA into your HDL project.

We said **almost normally** because there is one exception: adding a new file (HDL code, constraint, IP, etc.) to the project using the GUI is not enough. You **must also add** the file name in one of Hog's list files as explained in the following.

Let's now suppose that you want to add a new file to the project and that this file is located in `repo/dir1/` and is called `file1.hdl`.

First of all, the new file (that is unknown to git) must be added to the repository:

² You will have to choose the *protocol* that works for you: ssh, https, or krb5. We used the `--recursive` option to automatically clone all the submodules included. In general a HDL repository may or may not include other submodules, but the Hog scripts are always included as submodules. If you have cloned the repository without the recursive options (or if that option does not work, we heard that it happens on Windows), you will have to go inside it and initialise the submodules `git submodule init` and update them `git submodule update`.

³ Alternatively, you can type `cd Top` (the Top folder is always present in a Hog handled HDL repository) and type `ls`: each directory in this path corresponds to a Vivado/Quartus project in the repository.


```
cd repo
git add ./dir1/file1.hdl
git commit -m "add new file file1.hdl"
git push
```

Now that the file is safely on git, we have to add it to the Hog project, so that if another developer clones the repository, as you did at the beginning of this guide, the file will appear in the project⁴.

This is a new file, unknown to Hog for now, and we want it to be included into the project the next time that we run the CreateProject script described above. To do this, you must add the file name and path of `file1.vhd` into a Hog list file. The list files are located in `repo/Top/project1/list/`. Let's assume that the list file you want to add your file to is `lib1.src`.

Open the file with a text editor and add the file name and path in a new line.

Now that the new file is included in a list file, you can close the Vivado/Quartus project and re-create it by typing `./Hog/CreateProject.sh <project name>` again.

Do you really have to do this every time you add a new file to the project? There is a quicker way. You can add the file with the GUI and **also** add the file to a `.src` list file. If you choose to do this, in Vivado, you have to choose the correct library when adding the file. The library must have the same name of the `.src` file to which you added the source file. In our example, the hdl file was added to a list file called `lib1.src`, so the library that you have to choose is `lib1`. You can select the library in the Vivado GUI from a drop-down menu when you add the file.

This procedure is valid for any kind of source file. If your file is a constraint file, just add it to a `.con` list file in the list directory, e.g. `repo/Top/project1/list/lib1.con`. If your file comes from a submodule in the repository, you have to add it in the proper `.sub` list file.

Renaming a file already in the project

If you need to rename or move a file, say from `path1/f1.hdl` to `path2/f2.hdl` do so and change the name in the proper list file accordingly. Don't forget to rename the file on git as well:

```
git mv path1/f1.hdl path2/f2.hdl
git commit -m "Renamed f1 into f2"
git push
```

What can go wrong?

If you do something wrong (e.g. you add a name of a non-existing file, create a list file with an invalid extension, etc.) you will get an error when you run the CreateProject script. In this case read Hog's error message and try to fix it. If you do something wrong with Vivado library, the error will at synthesis time because Vivado will not be able to find the component.

⁴ Not all the files stored in the git repository are part of a project: there can be non hdl files, obsolete files that are stored just in case, new files that are not ready to be used. Moreover some files could be part of a project and not of another. In our example, the repository could contain `project2` and `project3` that use different subsets of files in the repository.

Adding a new IP

If you want to add a new IP core, say it's called **my_ip1**, you must create it in out-of-context mode and save the .xci file (and only that) in the repository in a subfolder of the special IP folder `repo/IP`.

Moreover, the xci file must be in a folder with the same name as the file, like this: `repo/IP/my_ip1/my_ip1.xci`.

If you want to keep different sets of IPs separate you can use additional subfolders in the IP directory, for example: `repo/IP/some_folder/my_ip1/my_ip1.xci`. Now you can add the .xci normally to any source list file in the list folder of your project.

When Vivado synthesises the IPs, it creates plenty of additional files where the .xci file is located. To avoid to commit those file to the repository, a .gitignore file is used. This file specifies to git that every file that is not a .xci file inside the IP directory must be ignored.

Vivado IP initialisation coefficient files (.coe)

If you have a .coe file for RAM initialisation, you cannot store it inside the IP folder, otherwise it will be ignored as explained earlier. You can store it anywhere else in the repository. Pay attention to specify the path to this file as a **relative path**. This must be done in the text box in vivado GUI when you customise the IP.

A couple of things before getting to work

Here you can find a couple of details and suggestions that can be useful when working with Hog-handled repository.

Commit before starting the workflow

All the Hog scripts handling version control are automatically added to your project: this means that you have the possibility to create a reproducible and traceable bitfile, even when you run locally. This will happen **only if you commit your local changes before running synthesis**. You don't have to push! Just commit locally, then you can push when you are sure that your work is good enough. If you don't commit, Hog will alert you with a Critical Warning at the beginning of the synthesis.

Different list files

As we have explained above, source files taken from different list files will be added to your project in different "libraries": the name of each library is the name of the list file. When working with components coming from different list files, you will need to formally include the libraries and call the component from the library it belongs to. For example, in VHDL:

```
library lib1
use lib1.all

...

u_1 : entity lib1.a_component_in_lib1
port map (
    clk => clk,
    din => din,
    dout => dout
);
```

If working within the same library, you can normally use the “work” library.

Wrapper scripts

A set of bash scripts can be used to run IP synthesis, project synthesis, implementation and bitstream generation without opening the vivado gui. The commands to launch are:

```
./Hog/LaunchSynthesis.sh <proj_name>
./Hog/LaunchIPSynth.sh <proj_name>
./Hog/LaunchImplementation.sh <proj_name>
```

These scripts call the Tcl scripts contained in `Hog/Tcl/launchers` that are used in the continuous integration. But as they work perfectly even locally, we wrapped them in these shell scripts so that you can use them locally if you don't want to open the GUI. Launching the implementation without having launched the synthesis beforehand will run all the previous stages, exactly as if you clicked the GUI button.

1.7.2 How to convert an existing project to Hog

Converting an existing project to Hog means: copying the source files into a git repository, adding Hog as a submodule, creating the Hog list files (text files containing the names of your source files), and creating a Tcl script able to create the Vivado/Quartus project.

Before starting

We will assume that you are starting from a clean repository and you want to convert a Vivado project stored in a local folder.

If you are migrating between two git repositories and you want to retain the history of your old repository have a look [here](#)

If you are migrating to Hog but you are not changing repository, you can follow the instructions below ignoring the creation of the new local repository. In this case you might want to work in a new branch of your repository.

Let's suppose your new repository called `new_repo` with url `new_repo_url`, your project is named `myproject` is currently stored in some local folder `mypath`. If you don't have a new repository you can go on Gitlab (gitlab.cern.ch) and create one.

Creating the git repository

First of all create a new folder where you will store your firmware, initialise it as a git repository and connect it to the remote repository:

```
mkdir new_repo
cd new_repo
new_repo> git init
new_repo> git remote add origin new_repo_url
```

For now we will work on the master branch, that is the default one. If you want you can create a branch and work on that:

```
git checkout -b first_branch
git push origin first_branch
```

Add Hog to your project

Hog repository should be included as a submodule in the root path of your repository. To do this type:

```
git submodule add ssh://git@gitlab.cern.ch:7999/hog/Hog.git
```

Here we assumed that you want to use the ssh protocol, if you want to use https enter

```
git submodule add https://gitlab.cern.ch/hog/Hog.git
```

If you like krb5:

```
git submodule add https://:@gitlab.cern.ch:8443/hog/Hog.git
```

However, it is good idea not to include the submodule protocol explicitly, but to let it be inherited from the repository Hog is included into. To obtain this edit a file called `.gitmodules` with your favourite text editor (say emacs):

```
emacs .gitmodules
```

In that file you should find a section as in the following. Modify the url value by replacing it with a relative path. Note that the url that must be specified relatively to the path of your repository:

```
[submodule "Hog"]
    path = Hog
    url = ../../hog/Hog.git
```

Now from your repository:

```
git submodule update
```

This should trigger an error if you made a mistake when editing the repository path.

Copying some templates from Hog and committing

Now that you have Hog locally, you can start setting up your repository. Hog provides a set of templates that you can use, you can add a `.gitignore`¹ to your repository with the following command:

```
cp Hog/Templates/gitignore .gitignore
```

Let's now make our first commit:

```
git add .gitignore .gitmodules Hog
git commit -m "Adding Hog to the repository"
git push origin master
```

If you are working in a branch that is not master, please replace the last instruction with:

¹ You might need to modify your `.gitignore` file if you want to do a more complicated directory structure, especially with the IP and BD files. For example, Hog template assumes that you store your IPs in `IP/ip_name/ip_name.xci`. If you do, this file would be enough for you. If you need a more complicated structure, you can edit the file or you can use several `.gitignore` files the subfolders of the main IP directory.

```
git push origin your_branch_name
```

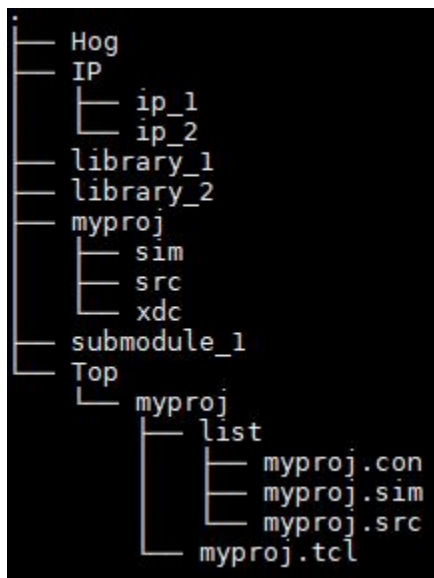
Early tagging your repository

Hog assumes that at least a version tag of the form **vM.m.p** is present in your repository. Let's now create the first Hog tag:

```
git tag v0.0.0
git push --tags
```

Generating Hog directory structure

You have now to generate a directory structure similar to this one:



A complete description of the meaning of each folder can be found in [THE *Hog User Manual*](#)

Top folder

Every Hog-handled HDL repository must have a directory called `Top`. In here, each subdirectory - that we call the **project top-direcotry** - represents a HDL project in the repository. You can start by creating your project top-directory:

```
mkdir -p Top/myproject
```

Every project top-directory, must contain a subdirectory called `list` where the so-called hog list file are stored. Let's create it:

```
mkdir Top/myproject/list
```

Moreover a tcl script, with the same name of the project (plus the `.tcl` extension) must be in the project top-directory. Hog runs this script, called the project tcl script, to create a project. This is a recap of what we have learned up to now:

- A `Top` folder must be in the repository
- Inside this folder there is one subfolder for each project in the repository, called the project top-directory

- Inside each project's top-directory there is 1. a `list` sub-directory containing: the list files of the project and 2. a tcl script used to create the project;

For more advanced Hog features, additional files and folder are located in the `Top` folder, but we don't discuss them now for simplicity.

In order to create the project's tcl script, we will start from the template provided in the `Hog/Templates` folder:

```
cp Hog/Templates/top.tcl Top/myproject/myproject.tcl
```

Use your favourite text editor to modify the template Tcl file. There are several things you can edit, depending on your needs, for example the FPGA name or the synthesis or implementation strategies. The template contains comments to guide you through the process.

Now you can commit everything you just did:

```
git add Top
git commit -m "Adding Hog Top folder"
```

Importing source files to the project

You are now ready to import the files needed to build your project².

First of all, copy the files from your local folder into the folder that contains the git repository. Exception made for some reserved directory (e.g. `Top`, `IP`, `BD`) you can put your files wherever you feel like inside your repository, organising them as you see fit.

In this example we will create a directory named `lib_myproject` where we will store all the source, simulation and constraint files.

```
mkdir -p lib_myproject/source lib_myproject/simulation lib_myproject/constraint
cp ../old_repo/source_1.vhd lib_myproject/source
cp ../old_repo/source_2.vhd lib_myproject/source
...
cp ../old_repo/simulation_1.vhd lib_myproject/simulation
cp ../old_repo/simulation_2.vhd lib_myproject/simulation
...
cp ../old_repo/constraint_1.vhd lib_myproject/constraint
cp ../old_repo/constraint_2.vhd lib_myproject/constraint
...
```

After having added all the relevant files in your folders you have to add their path and file names to the appropriate list files. In this example, we will create:

- One source list-file called `Top/myproject/list/myproject.src`, containing the source files of your project
- One simulation list-file called `Top/myproject/list/myproject.sim`, containing the files used in the simulation (e.g. test benches, modules that read/write files, etc.)
- One constraint list-file called `Top/myproject/list/myproject.con`, containing your constraints (.xdc, .tcl, etc.)

You can copy and modify this bash script to ease this quite tedious part of the work:

² Hog gives you the possibility to organise the source files in different VHDL libraries (Verilog doesn't have the concept of library). You can add your source files into several .src files in the list directory, each of these .src files will correspond to a different library with the same name as the .src file (excluding the .src extension). For simplicity, in this chapter we will assume the presence of a unique library with the same name of your project.

```

for i in $( ls lib_myproject/source/* ); do \
    echo $i >> Top/myproject/list/myproject.src; \
done
for i in $( ls lib_myproject/simulation/* ); do \
    echo $i >> Top/myproject/list/myproject.sim; \
done
for i in $( ls lib_myproject/constraint/* ); do \
    echo $i >> Top/myproject/list/myproject.con; \
done

```

Note that the path of the file is specified with respect to the main folder of the repository.

If you want, you can add comment lines in the list-files starting with a # and you can leave empty lines (or lines containing an arbitrary number of spaces). All of these will be ignored by Hog.

At this point, you might want to check that the files are correctly picked up by regenerating the Hog project: `./Hog/CreateProject.sh myproject`, Hog will give you an error if a file is not found. You can open the created project in `VivadoProject/myproject/myproject.xpr` or `QuartusProject/myproject/myproject.qpf` with the GUI and check that all the files are there. If not, modify the list files and create the project again. When you are satisfied, you can commit your work:

```

git add lib_myproject
git add Top/myproject/list/myproject.src
git add Top/myproject/list/myproject.sim
git add Top/myproject/list/myproject.con
git commit -m "Adding source files"

```

Submodules

If your project uses source or simulation files hosted in a separate repository you can add that repository as a git submodule.

```
git my_submodule add my_submodule_url
```

You must add all your submodules in the root directory of your repository.

Files taken from a submodule must be added to a special list-file having the `.sub` extension. Moreover the name of the file must be the same of the submodule directory³.

Add the relevant source files to the submodule list-file. You can copy and modify the following script if you want:

```

for i in $( ls submodule/* ); do \
    echo $i >> Top/myproject/list/my_submodule.sub; \
done

```

Now commit the newly created `.sub` file:

```

git add Top/myproject/list/my_submodule.sub
git commit -m "Add a new my submodule"

```

³ In case this naming limitations complicate your work too much, please note that the submodule folder name can differ from the submodule url.

IP files

IP files must go in a special folder called `IP` in the root of your repository. The `IP` directory can contain all the subdirectories you want, but there is a rule: each ip file (.xci for Vivado) must be contained in a sub-folder called with the same name as the .xci file (extension excluded).

Basically for each IP in your project run:

```
mkdir -p IP/ip_name/  
cp ../old_repo/ip_name.xci IP/ip_name/
```

Then you can add the xci files to the .src list file you want, in this case we will use a separate file called `IP.src`⁴. You can use the following script if you like:

```
for i in $( ls IP/* ); do \  
    echo $i/$i.xci >> Top/myproject/list/myproject.src;  
done
```

As usual, you can check that the files are correctly picked up by regenerating the project `./Hog/CreateProject.sh myproject`. If you are satisfied with the changes, you can commit your work.

```
git add IP  
git add Top/myproject/list/IP.src  
git commit -m "Adding IP Files"
```

IP initialization files (.coe)

Please note that the `.gitignore` template provided by Hog adds constraints on the `IP` folder. Out of all the files contained in `repo/IP/`, git will pick up only .xci files. Files with different extensions will be ignored. If you have .coe files for RAM initialization or analogous files please make sure that you store these files in a separate folder and point to them in the `IP` one by using a relative path.

The top file of your project

Your project must contain a module called `top_myproject`, i.e. your project's name preceded by `top_`. Hog will pick up such module and set it as the Top of your project.

Since Hog will back annotate your project to track the source code used in each build, extra generics will be provided to your top file at the beginning of the synthesis. You can add the following generics to your top file in order to store those values in some registers:

```
generic (  
    -- Global Generic Variables  
    GLOBAL_FWDATE      : std_logic_vector(31 downto 0);  
    GLOBAL_FWTIME      : std_logic_vector(31 downto 0);  
    TOP_FWVERSION      : std_logic_vector(31 downto 0);  
    TOP_FWHASH         : std_logic_vector(31 downto 0);  
    XML_VERSION        : std_logic_vector(31 downto 0);  
    XML_HASH           : std_logic_vector(31 downto 0);  
    GLOBAL_FWVERSION   : std_logic_vector(31 downto 0);  
    GLOBAL_FWHASH      : std_logic_vector(31 downto 0);  
    XML_HASH           : std_logic_vector(31 downto 0);
```

(continues on next page)

⁴ There is no concept of library for the IPs, so we prefer to put them in a separate .src file. You can put them in the same list file as your other source files if you wish. Just open `Top/myproject/list/myproject.src` with a text editor and add them there.

(continued from previous page)

```

XML_VERSION          : std_logic_vector(31 downto 0);

HOG_HASH             : std_logic_vector(31 downto 0);
HOG_VERSION          : std_logic_vector(31 downto 0);

-- Project Specific Lists (One for each .src file in your Top/myproj/list folder)
<MYLIB0>_FWVERSION    : std_logic_vector(31 downto 0);
<MYLIB0>_FWHASH       : std_logic_vector(31 downto 0);
<MYLIB1>_FWVERSION    : std_logic_vector(31 downto 0);
<MYLIB1>_FWHASH       : std_logic_vector(31 downto 0);

-- Submodule Specific variables (only if you have a submodule, one per submodule)
<MYSUBMODULE0>_FWHASH : std_logic_vector(31 downto 0);
<MYSUBMODULE1>_FWHASH : std_logic_vector(31 downto 0);

-- External library specific variables (only if you have an external library)
<MYEXTLIB>_FWHASH     : std_logic_vector(31 downto 0);

-- Project flavour
FLAVOUR               : integer
);

```

In our case, there is only one library called myproject, possibly a submodule, and no XML_VERSION, flavour or external libs. These are more advanced Hog features that are not treated in this guide.

All your source files are now compiled as a separate library called according to the .src file they are contained in. So if you are using multiple .src files, you have to add the library to your project:

```

library myproject;
use myproject.all;

```

If you are using a module that is included in a different .src file with respect to your top module, you will have to specify the library when you use it:

```

u_1 : entity myproject.a_component_in_lib1
port map (
  clk => clk,
  din => din,
  dout => dout
);

```

If you work within the same library (i.e. .src file) you can normally use the work library.

Create your project

As you know, if you run `./Hog/CreateProject.sh myproject`, Hog will create your project in `VivadoProject/myproject/myproject.xpr` or `QuartusProject/myproject/myproject.qpf`. You can open the project with the GUI and check that everything looks alright. The `.gitignore` template, that you copied in your repository, takes care of ignoring the VivadoProject directory (or QuartusProject) as we do not want to commit Vivado/Quartus files to the repository. If you decided not to use the provided template you should take care of ignoring this directory. Also you should try to run the complete workflow as something might have gone wrong with your IPs or the libraries. If something is indeed wrong, try to fix it by modifying: the source files, the list files, the project tcl file. If you modify the list files or the project tcl file, you have to re create the project to see if the modifications had the desired effect.

Code documentation

Hog can be used to automatically generate and deploy documentation for your code. Hog works with [Doxygen](#) version 1.8.13 or later. If your code already uses Doxygen style comments, you can easily generate Doxygen documentation. You just have to create a directory named `doxygen` containing the files used to generate the HDL documentation. A file named `doxygen.conf` should be in this directory together with all the files needed to generate your Doxygen documentation. You can copy a template configuration from `Hog/Templates/doxygen.conf` and modify it.

Wrapper scripts

There are three scripts that can be used to run synthesis, implementation and bitstream generation without opening the Vivado GUI. The commands to launch them are

```
./Hog/LaunchSynthesis.sh <proj_name>
./Hog/LaunchImplementation.sh <proj_name>
./Hog/LaunchWriteBistream.sh <proj_name>
```

Launching the implementation or the bistream generation without having launched the synthesis beforehand will run the synthesis stage too.

1.7.3 How to update Hog to a new release

This guide will help you to Update Hog to the version you want. We assume you already have a repository handled with Hog that is hosted on Cern Gitlab.

On your project's Gitlab website, create a new merge request and a new branch starting from master¹.

Now if you want to update Hog to the latest version:

```
cd Hog
git checkout master
git pull
```

Now do `git describe` to find out what version you got:

```
git describe
```

You should obtain something like `HogYYYY.n` (e.g. `Hog2020.1`) or `vX.Y.Z` (e.g. `v1.2.3`).

If you want to update to a specific version, go to Hog [repository][gitlab.cern.ch/hog/Hog] and choose what version you want to update to, let's say you pick version `vX.Y.Z`.

```
cd Hog
git checkout vX.Y.Z
```

Now you have to update your `.gitlab-ci.yml` file. This is the file used to configure Gitlab continuous integration (CI), if you are not using Gitlab CI -hence the file is not be there- you can skip this part. Go back to your repo and edit the `.gitlab-ci.yml` with your favourite editor, say `emacs`:

```
cd ..
emacs .gitlab-ci.yml
```

¹ If you are already working on a branch on your HDL repository, you can update Hog within that branch. (You can also create a new branch locally and open the merge request on the website later.) Checkout the branch and go into the Hog directory.

At the beginning, in the include section you have a ref, similar to this:

```
include:
- project: 'hog/Hog'
  file: '/hog.yml'
  ref: 'va.b.c'
```

In older Hog versions, the file used to be called `gitlab-ci.yml`, so if you were using an older version of Hog, please change `gitlab-ci.yml` to `hog.yml`. Change the ref to the new version you've just pulled:

```
include:
- project: 'hog/Hog'
  file: '/hog.yml'
  ref: 'vX.Y.Z'
```

Save the file close the editor.

Now you kist have to commit to git and push the modification you have made. Add the modified files (Hog submodule and `.gitlab-ci.yml`), commit, and push:

```
git add .gitlab-ci.yml Hog
git commit -m "Update Hog to vX.Y.Z"
git push
```

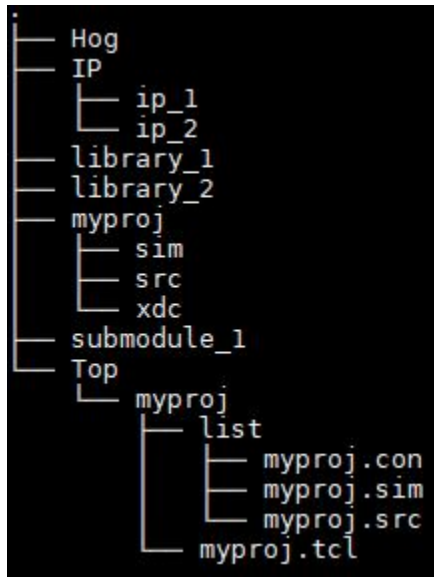
Finally go to your project's Gitlab website, open a merge request for your branch (if you have not done already) and merge it.

1.7.4 Hog general overview

In this section we will describe how a project is set-up using Hog locally and for the Continuous Integration (CI). In this section we will make no assumptions on the code you already have. If you already have a repository with code in it, you can also refer to this guide: [How to convert existing project to hog](#) section.

Project directory structure

Hog relies on some fixed directories and file names in the HDL repository. A typical directory structure is shown in figure:



HDL source files, together with constraint files and simulation files can be located anywhere in the repository. In the example shown in the figure above, they are located in the `myproj` directory and, possibly, in the `library_1` and `library_2` directories.

A Hog-handled repository can contain multiple (Vivado/Quartus) projects, each of them corresponding to a subdirectory of the **Top** folder. Each one of these subfolders, is referred to as the “top project-directory”. In the figure above we have just one project called **myproj**. Each top project-directory **must** contain all of the following:

- A *tcl file* that **must** have the same name as top project-folder (plus the `.tcl` extension). We will call this file the “project tcl file” or “project tcl script”.
- A *list subdirectory* containing the so-called list files i.e. special text files containing the names of the source files to be included in the project.

In figure, the project is called **myproj** hence the project tcl file is called **myproj.tcl**.

When you create the project, running the `Hog/CreateProject.sh`, Hog runs the project tcl file using Vivado or Quartus and creates the complete project in another directory, in our case `VivadoProject/myproj` or `QuartusProject/myproj`. In this directory you will find the typical Vivado or Quartus file structure.

Multiple projects or multiple repositories?

As is now clear, a Hog-handled repository can contain multiple projects. You may wonder when it is the case of having many projects in the same repository and when you want to create different repositories.

Using different projects in the same git repository

You should use many projects in the same repository when they share a significant amount of code. In this case, the version of the repository will be shared among all the projects. This is meaningful if the projects are strongly interconnected and it is unlikely for a project to change version without any modification to the others.

A typical use case is when the projects are intended for different devices (FPGAs) mounted on the same board.

To keep different parts of the project conceptually separated, it is possible to use many **libraries** as explained in the following. In this case, Hog evaluates the version (and the SHA) independently for each library, so it is possible to tell at a glance if two binary files share the same library.

For example you can have an FPGA with a “infrastructure” library containing all the circuitry to handle communication with the external world, and an “algorithm” library, containing the actual part of the design that processes data. Hog libraries will allow you to tell if two different binary files are generated using exactly the same source code for the algorithm but they have a different infrastructure.

Using different git repositories

If you don’t have any code sharing between two HDL projects, or if the shared code is minimal, you may think of having different repositories.

In this case, everything will be decoupled, as the two repository are two completely unlinked things. All that is explained in this guide will have to be done with both repositories and you can also, in principle, use two different versions of Hog.

In case you have a shared part of the code, in order to avoid code repetition, you can include the shared code as a git submodule. This must be a third git repository, also independent from the previous two.

If the code contained in the submodule is not meant to be working stand alone, it is not necessary to include Hog in it.

Hog directory

The Hog repository should be included as a submodule into your HDL repository, following these rules:

1. Hog must be in the root path of your repository
2. The directory name must be “Hog”

Moreover it is recommended not to include the submodule protocol explicitly, it is much better to inherit it from the repository Hog is included into.

To obtain this you can run the following commands in the root folder of your repository.

```
git submodule add <protocol>://gitlab.cern.ch/Hog/Hog.git
git config --file=.gitmodules submodule.Hog.url ../Hog/Hog.git
git submodule sync
git submodule update --init --recursive --remote
```

Remember to chose your protocol among ssh, https, git, or krb5. Also note that `../Hog/Hog.git` must be replaced with the correct path, relative to your repository. A git error will be generated by `git submodule update --init --recursive --remote` if the path is not properly set. Alternatively you could add the submodule normally and then edit the `.gitmodules` file, as explained [here](#).

Source directories

Source directories contain HDL files used for synthesis. HDL files can be placed anywhere in your repository, but it is advised to arrange them according to their library¹. We suggest to put HDL files belonging to separate libraries in separate folders although this is not mandatory. The exact structure or name of this folder is not enforced.

Top level entity

The top module of your project **must** be called `top_<project_name>`. This module can be contained in any file stored anywhere in the repository as long as it is linked in a *.src file*.

Hog extracts repository information (git commit 7-digit SHA and numeric version stored in git tags) and feeds the resulting values to the design using VHDL generics or Verilog parameters. A full list of these can be found in the *Hog generics* section. A template for the top level file (in VHDL and Verilog) is available in the *Hog/Template* directory. A full description of the templates can be found in the *available templates* section.

Git submodules

Hog is designed to handle git submodules, i.e. if you use some code contained in a git repository you can simply add it to your project using:

```
git submodule add <submodule_url>
```

All the submodules **must** be placed in the root directory of your repository. Suppose that you have 2 submodules called *sub_1* and *sub_2*:

```
Repo/sub_1  
Repo/sub_2
```

When you add files contained in a submodules you have to use the *.sub* list files described *here*.

Top directory

The **Top** directory must be located in the root folder of the repository:

```
Repo/Top
```

It contains one directory for each of your projects (say *proj_1*, *proj_2*, *proj_3*):

```
Repo/Top/proj_1  
Repo/Top/proj_2  
Repo/Top/proj_3
```

As previously mentioned, these 3 directories are called the “top project-directories”.

¹ The concept of library does not exist in Verilog and SystemVerilog

Top project-directory

Each of the project directories must contain the tcl file that generates the project. The .tcl file contained in the project directory must contain the instructions to build your project. They must be named as follows:

```
Repo/Top/<project_name>/<project_name>.tcl
```

To trigger all Hog functionalities, the last line of the tcl script must be:

```
source $PATH_REPO/Hog/Tcl/create_project.tcl
```

A template for the <project_name>.tcl file is available in the [Hog/Template](#) directory. A full description of the template can be found in the [available templates](#) section. More information on the tcl script can be found in the [project tcl file](#) section.

If you want some custom operation to be performed before the project creation (e.g. you want create a source file using a Tcl script), you can insert your instruction before this line. If you want some custom operation to be performed after the project is created, you can add the Tcl instruction after the `create_project.tcl` call. Do this at your own risk.

List directory

A directory named *list* must be in each of the top project-folders. This directory contains the list files, that are plain text files, used to instruct Hog on how to build your project. Each list file contains the names to be added to the *proj_1* project. Hog uses different kinds of list files, identified by their extension:

- `.src`: used to include HDL files belonging to the same library
- `.sub`: used to include HDL files belonging to a git submodule
- `.sim`: used to include files use for simulation of the same library
- `.con`: used to include constraint files
- `.prop`: used to set some Vivado properties, such as the number of threads used to build the project.
- `.ext`: used to include HDL files belonging to an external library

In .src, .sub, .sim, and .con list files, you must use paths relative to the repository location to the files to be included in the project.

.ext list file must use absolute paths. To use the firmware Continuous Integration this path must be accessible to the machine performing the git CI, e.g. can be on a protected afs folder.

More information on the list file can be found in the dedicated [list files](#) section.

IP directory

All the IPs xci files **must** be stored in the *repo/IP/* repository. To add a new IP core, that must be created in out-of-context mode. The .xci file (and only that one!) must be saved and committed to in the repository in *repo/IP/ip_name/ip_name.xci*. Please note that the name of the folder must be the same as the xci file. Now you can add the .xci normally to any .src list file in the list folder of your project.

IP initialisation files (.coe)

Please note that the `.gitignore` template provided by Hog adds constraints on the IP folder. Out of all the files contained in *repo/IP/*, git will pick up only xci files. Files with different extensions will be ignored. If you have .coe files for RAM initialisation or analogous files please make sure to store these files in a separate folder and point to them in the IP directory by using a relative path.

Auto-generated directories

The following directories are generated at different stages of library compilation or synthesis/implementation time. These directories should never be committed to the repository, for this reason they are listed in the `.gitignore` file. You can always delete any of these directories with no big consequences: they can always be regenerated by Vivado/Quartus or Hog scripts.

VivadoProject or QuartusProject

When you generate a project with Hog, it will create a sub-directory here. When everything is generated, this directory contains one subdirectory for each project in the repository, containing the Vivado (Quartus) project-file `.xpr` (`.qpf`). The name of the sub-directory and of the project file are always matching. In our case:

```
Repo/VivadoProject/proj_1/proj_1.xpr
Repo/VivadoProject/proj_2/proj_2.xpr
Repo/VivadoProject/proj_3/proj_3.xpr
```

The *Repo/VivadoProjects/proj_3/* directory also contains Vivado automatically generated files, among which the Runs directory:

```
Repo/VivadoProjects/proj_1/proj_1.runs/
```

That contains one sub-folder for every Vivado run with all the IPs in your project, the default Vivado synthesis run (`synth_1`) and implementation run (`impl_1`). Hog will also copy IPbus XMLs and generated binary files into *Repo/VivadoProjects/proj_1/proj_1.runs/* at synthesis/implementation time.

SimulationLib

Modelsim or Questasim compiled libraries will be placed here and automatically linked to your project. The library compilation can be done automatically if the `vsim` executable is in your PATH and you launch the `hog/Init.sh` script.

Optional directories

Doxygen

The `doxygen` directory contains the files used to generate the HDL documentation. A file named *doxygen.conf* should be in this directory, together with all the files needed to generate your Doxygen documentation. VHDL is well supported with Doxygen version 1.8.13 or later, so Hog will not use any older version.

Wrapper scripts

There are launcher scripts in the Hog directory that can be used to run simulation synthesis, implementation and bitstream writing without opening the Vivado GUI. The commands to launch them are

```
./Hog/LaunchSimulation.sh <proj_name>
./Hog/LaunchSynthesis.sh <proj_name>
./Hog/LaunchImplementation.sh <proj_name>
./Hog/LaunchWriteBistream.sh <proj_name>
```

Launching the implementation will run the synthesis and IP synthesis stages too if not previously launched. More information on these scripts can be found in the dedicated [How to create and build project](#) section.

1.7.5 Hog Local

Project Tcl file

As previously stated Hog uses a TCL script located in `./Top/<my_project>/<my_project>.tcl` to generate the HDL project.

The `<my_project>.tcl` is expected to define few basic variables containing the information needed to build your project. The tcl script is expected to call the `./Hog/Tcl/create_project.tcl` script after setting the needed environment variables. The latter script will read back the variables and generate the HDL project. This section contains a full recipe to build the tcl script for your project.

A template for a Vivado project can be found under `./Hog/Templates/top.tcl`.

Telling Hog the HDL compiler to be used

The first line of your tcl script is expected to indicate Hog which HDL compiler to be used to generate your project. To do this the first line in the tcl script file must be a comment containing the name of the tool to be used. The following tools are recognised:

- `#vivado`
- `#vivadoHLS`
- `#quartus`
- `#intelHLS`

If this line is not available Hog will assume your project runs under Vivado.

Note: ‘vivadoHLS’ and ‘quartus’ options are currently under development. If you are willing to use the corresponding feature branch. Note that no support is guaranteed.

‘intelHLS’ is not supported will simply return an error message.

TCL Variables

The `./Hog/Tcl/create_project.tcl` uses the following variables to build your project.

FPGA

The FPGA variable indicates the target device code. This variable is *mandatory*. It must be chosen among the ones provided by the chosen HDL compiler. As an example for a Xilinx Virtex-7 FPGA it could be set to `xc7vx330tffg1157-2`. Note that the exact code will depend on the full characteristics of the device you are using, e.g. number of logic cells, package, speed grade, etc.

FAMILY

The FPGA variable indicates the device family. This variable applies to *Quartus only*. The value must be chosen among the ones provided by the chosen HDL compiler. As an example for a Intel MAX10 FPGA it must be set to `"MAX 10"`. *NOTE* that the variable value is included in quotation marks.

SYNTH_STRATEGY

The SYNTH_STRATEGY variable indicates the synthesis strategy to be used. It has to be chosen among the ones provided by the chosen HDL compiler. As an example for Vivado you could use: `"Vivado Synthesis Defaults"`. *NOTE* that the variable value is included in quotation marks.

SYNTH_FLOW

The SYNTH_FLOW variable indicates the synthesis flow to be used. It has to be chosen among the ones provided by the chosen HDL compiler. As an example for Vivado you could use: `"Vivado Synthesis 2019"`. *NOTE* that the variable value is included in quotation marks.

IMPL_STRATEGY

The IMPL_STRATEGY variable indicates the implementation strategy to be used. It has to be chosen among the ones provided by the chosen HDL compiler. As an example for Vivado you could use: `"Vivado Implementation Defaults"` or `"Performance_Retiming"`. *NOTE* that the variable value is included in quotation marks.

SIMULATOR

The simulation software used to run the *simulation sets* in your project. Possible values are `"questa"`, `"modelsim"`, `"xsim"`. These strings are case-insensitive.

IMPL_FLOW

The IMPL_FLOW variable indicates the implementation flow to be used. It has to be chosen among the ones provided by the chosen HDL compiler. As an example for Vivado you could use: “Vivado Implementation 2019”. *NOTE* that the variable value is included in quotation marks.

DESIGN

The DESIGN variable indicates the name of your project. This variable is *mandatory*. It must be automatically set in the tcl file, i.e. use “[file rootname [file tail [info script]]]” to get the name of the <my_project>.tcl script. *NOTE* that the variable value is included in quotation marks.

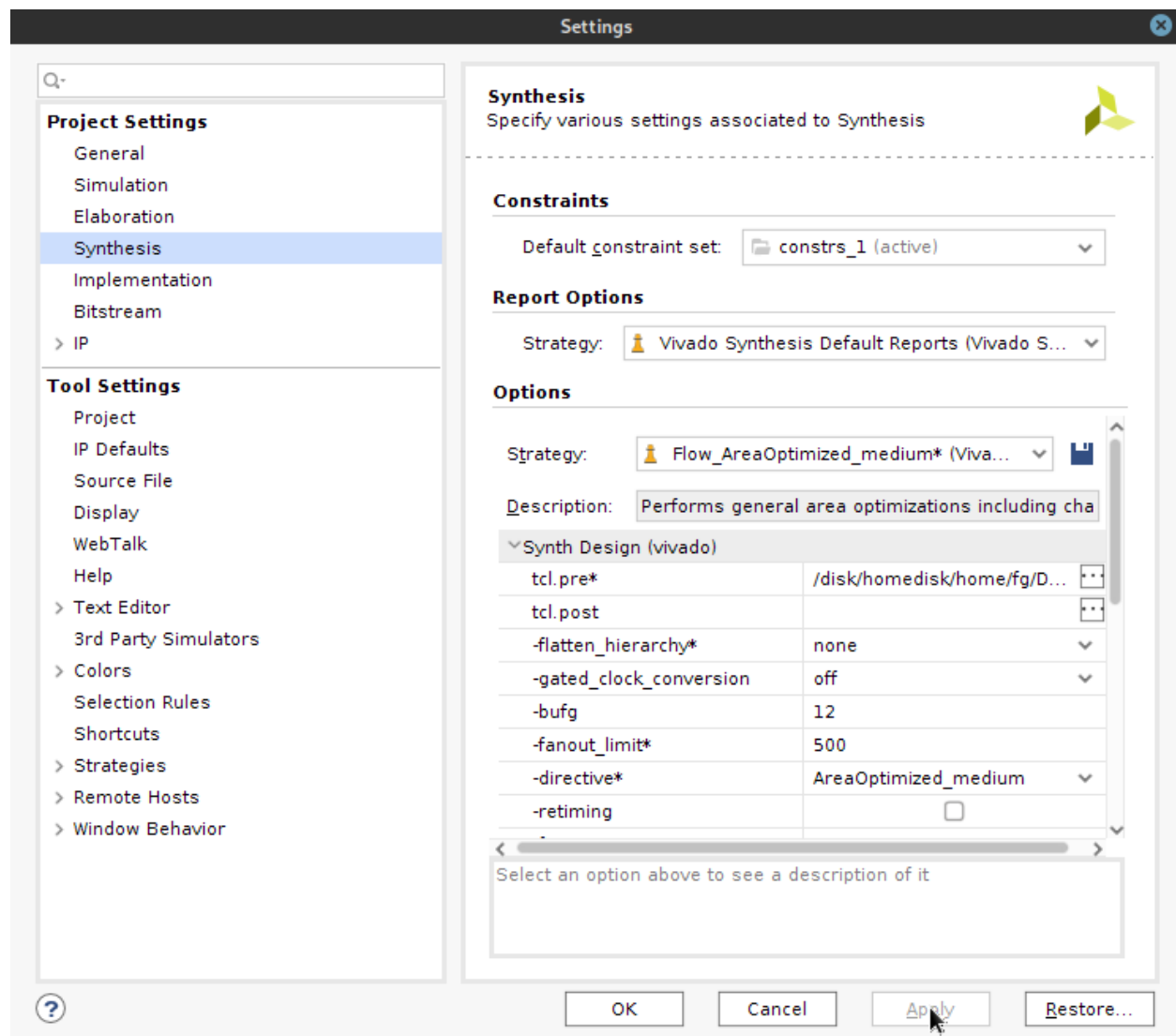
PROPERTIES

The PROPERTIES variable allows you to add optional additional properties to be set while creating your project. This variable is optional and can be left empty.

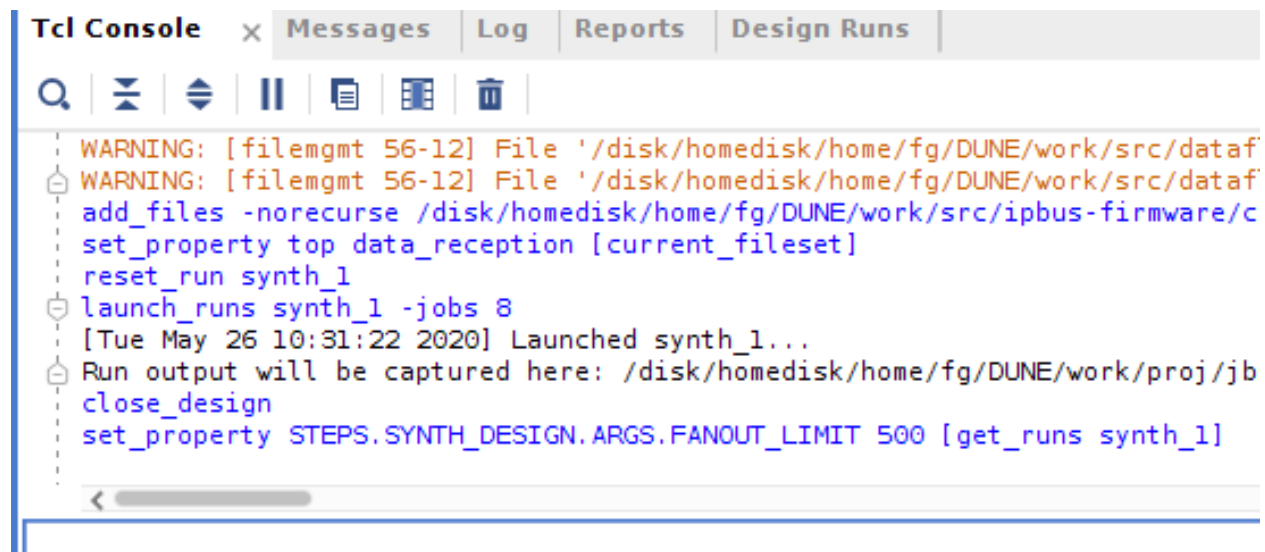
NOTE: you should use one line per property and that the ‘ ’ character must be the last one of each line

To set a property you must define two dictionaries, one for synthesis and one for implementation. The dictionaries must have the names of the corresponding Vivado runs. The default Vivado run names are: synth_1 for synthesis and impl_1 for implementation.

To find out the exact name and value of the property, use Vivado GUI to click on the checkbox you like.



This will make Vivado run the `set_property` command in the Tcl console.



Then copy and paste the name and the values from the Vivado Tcl console into the lines below.

An example of properties setting is:

```
set PROPERTIES [dict create \
    synth_1 [dict create \
        STEPS.SYNTH_DESIGN.ARGS.FANOUT_LIMIT 600 \
        STEPS.SYNTH_DESIGN.ARGS.RETIMING true \
    ] \
    impl_1 [dict create \
        STEPS.OPT_DESIGN.ARGS.DIRECTIVE Default \
    ] \
]
```

PATH_REPO

The PATH_REPO variable indicates the path to the root folder of your repository. This variable is *mandatory*. The value must be set automatically in the tcl script, i.e. use “[file normalize [file dirname [info script]]]/../”

BIN_FILE

The PATH_REPO variable indicates the output extension for the output file. If this variable is set to ‘1’, the implementation will create a binary file (.bin) containing only device programming data, without the header information found in the standard bitstream file (.bit). This variable is optional and its default value is ‘0’.

Running additional scripts

The <my_project>.tcl script can source other additional scripts contained in your repository. If you wish to run some scripts before creating your project then place them before calling ./Hog/Tcl/create_project.tcl. This can be used to generate additional files to be included in your project.

The ./Hog/Tcl/create_project.tcl will finish leaving you project open, you can run additional scripts on your project by placing them after ./Hog/Tcl/create_project.tcl.

Warning: do this at your own risk.

List Files

A directory named *list* must be in each of the top project-folders. This directory contains the list files, that are plain text files, used to instruct Hog on how to build your project. Each list file shall contain the list of the files to be added to the *proj_1* project.

Properties can also be specified in the list files, adding them after the file name, separated by any number of spaces. Comments can be added using #. A generic list file looks therefore like this,

```
source_dir/file1.vhd [prop_1] [prop_2]
# comment here, for example
source_dir/file2.vhd [prop_3]
source_dir/file3.vhd
```

Hog uses different kinds of list files, identified by their extension:

- `.src` : used to include HDL files belonging to the same library
- `.sub` : used to include HDL files belonging to a git submodule
- `.sim` : used to include files use for simulation of the same library
- `.con` : used to include constraint files
- `.prop` : used to set some Vivado properties, such as the number of threads used to build the project.
- `.ext` : used to include HDL files belonging to an external library

In `.src`, `.sub`, `.sim`, and `.con` list files, you must use paths relative to the repository location to the files to be included in the project.

.ext list file must use absolute paths. To use the Hog CI, this path must be accessible to the machine performing the git CI, e.g. can be on a protected afs folder.

List files are recursive

A list file can contain another list file, whose content will be then included.

You can include a mixture of source files and list files, Hog will recognise them by the extension.

This feature can be used if multiple projects on the same repository share a subset of files. A common list file can be created and stored anywhere in the repository and then included in the specific project list files.

Source list files (.src)

Hog creates a HDL library¹ for each `.src` list file and assign all the file included in the list file to that library. Filibrarieses with the extension are used to include HDL files belonging to a single library and the `.xci` files of the IPs used in the library.

For example, if there is a `lib_1.src` file in the list directory like this:

```
source_dir/file1.vhd
source_dir/file2.vhd
source_dir/file3.vhd
IP/ip1.xci
IP/ip2.xci
```

the files will be included into the Vivado project in the `lib_1` library. Note that if you include another `.src` list in your `.src` file, this will not be included in the library but will take the library name from the original `.src` file.

To use them in VHDL² you should use the following syntax:

```
library lib_1
use lib_1.all

...

u_1 : entity lib_1.a_component_in_lib1
port map (
  clk => clk,
  din => din,
  dout => dout
):
```

¹ https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_working_with_vhdl_libraries.htm

² Libraries are ignored in Verilog and SystemVerilog

The following properties can be specified for files in a `.src` list:

- 93: (only for `.vhd` and `.vhdl` files) File type is VHDL 93. If not specified Hog will use VHDL2008.
- XDC: File Type is XDC
- nosynth: File will not be used in synthesis
- noimpl: File will not be used in implementation
- nosim: File will not be used in simulation

Submodule list files (.sub)

To add files from a submodule to your project you must list them in a `.sub` list file. This tells Hog that those files are taken from a submodule rather than from a library belonging to the main HDL repository. Hog will not try to evaluate the version of those files, but it will evaluate the git SHA of the submodule.

Note: The list file must be called as the submodule itself.

For example, if you include the submodule `repo/sub_1/` then the corresponding list file must be `repo/Top/proj/list/sub_1.sub`

The same properties as for the `.src` list can be specified.

Simulation list files (.sim)

In this files are listed all the HDL files used only for simulations These include test benches and other files that are not used in synthesis: for example modules used to read and write from and to files.

For each `.sim` file, Hog creates a so-called “simulation set”.

For more information about simulation and `.sim` simulation list-files see the *specific chapter*

Constraint list files (.con)

All constraint files must be included by adding them into the `.con` files. Both `.xdc` (for Vivado) and `.tcl` files can be added here. The `nosynth` and `noimpl` properties can be specified here, if required.

For example, a `.con` file looks like:

```
constr_source_dir/constr1.xcf      #constraint used for synthesis and implementation
constr_source_dir/constr2.xcf nosynth      #constraint not used in synthesis
constr_source_dir/constr3.xcf noimpl      #constraint used in synthesis only
```

Properties list files (.prop)

Properties list files (.prop) are used to collect a small and optional set of Vivado properties.

Currently supported properties:

- `maxThreads <N>`: sets the number of threads used to build the project. Multi-threading is disabled by default to improve the reproducibility of firmware builds.

External proprietary files (.ext)

External proprietary files that are protected by copyright and cannot be published on the repository shall be included using the *.ext list file. **.ext list files must use an absolute path.** To be able to use the firmware CI, this path must be accessible to the machine performing the git CI, e.g. can be on a protected afs folder. This procedure has to be used **ONLY** in the exceptional case of files that can not be published because of copyright.

The *.ext list file has a special syntax since the md5 hash of each file must be added after the file name, separated by one or more spaces:

```
/afs/cern.ch/project/p/project/restricted/file_1.vhd 725cda057150d688c7970cfc53dc6db6  
/afs/cern.ch/project/p/project/restricted/file_2.xci c15f520db4bdef24f976cb459b1a5421
```

The md5 hash can be obtained by running the md5sum command on a bash shell

```
md5sum <filename>
```

the same checksum can be obtained on the Vivado or QuartusPrime tcl shell by using:

```
md5::md5 -filename <file name>
```

Hog, at synthesis time, checks that all the files are there and that their md5 hash matches the one contained in the list file.

Simulation

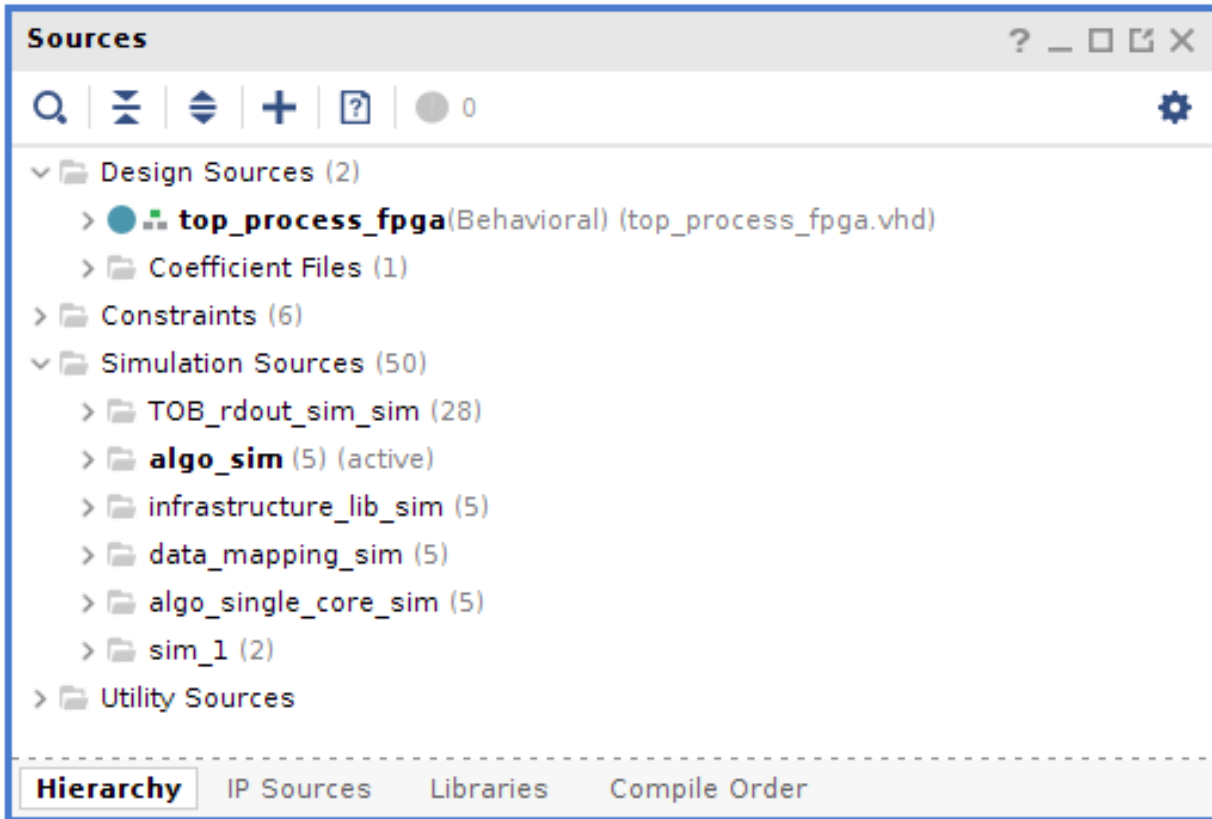
Hog supports Vivado Simulator (also called Xsim), Mentor Graphics Questasim and Mentor Graphics Modelsim.

The simulation setup is always handled by Vivado/Quartus, as if the simulation button on the GUI was clicked, then the simulation run is done with the simulation software.

Simulation sets

Simulations are organised in so-called “simulation sets” as in Vivado.

PROJECT MANAGER - process_fpga.1



In this figure you can see an example of simulation sets, these are contained in Vivado in the `Simulation_Sources` folder in the hierarchy view. You can right click on each of them, set it as active (becomes bold) and run it by clicking the “Run simulation button”. In the figure above `algo_sim` is the active simulation.

Every simulation set has a top entity, the test bench containing the device under test, and can be simulated independently.

In Hog every simulation set correspond to a `.sim list file`. In the `.sim list-files`, in addition to the properties defined for the `.src list files`, some other properties are specified. These properties are:

- `topsim=<entity_name>`: (mandatory) The name of the entity that will be the top level of the simulation;
- `wavefile`: (optional) it indicates the name of the entity you want to set as top level in your simulation (Questasim/Modelsim only);
- `dofile`: (optional) it indicates the file containing the signal waveforms to be observed in your simulation (Questasim/Modelsim only).

Project simulator

The software used to simulate is set at project level in Vivado¹. To change it you have to change the `SIMULATOR` parameter in your *project Tcl file*.

Simulation in the CI

If the simulation job configured in your projects CI, the *Hog CI* will automatically run the simulation sets.

The CI stage will fail if one simulation fails. This means that, in order for the CI stage to be meaningful, each simulation set should be designed to fail if some unwanted behaviour is present.

Even if Vivado doesn't allow for different simulators in the same project, in Hog CI it is possible to specify a different software to be used in each set. This can be done by adding at the top of the `.sim` list one of the following lines:

```
#Simulator xsim      # For Vivado Simulator
#Simulator questa    # For QuestaSim Simulator
#Simulator modelsim  # For ModelSim Simulator

#Simulator skip_simulation #To skip the simulation in the CI
```

If you do not specify the simulator software, Hog will set ModelSim as default.

If for any reason you don't want a specific simulation set to be simulated by the CI, just use `skip_simulation`.

An example `.sim` list file looks like this

```
#Simulator Xsim
tb_source_dir/tb_for_lib1.vhd topsim=tb_lib1
wave_source_dir/wave_lib1.tcl wavefile
do_source_dir/dofile_lib1.do dofile
tb_source_dir/another_file.vhd
```

Simulating without GUI

If you want to run the simulation locally, but without using the GUI (i.e. in the same way that CI does), you can use the `Hog/LaunchSimulation.sh` wrapper script.

```
Hog/LaunchSimulation.sh <proj_name> [library path]
```

This script will launch all the simulation sets in the specified projects, using the software specified in the `.sim` file and skipping the simulation if `skip_simulation` is specified.

This script will assume that the Modeslim/Questasim libraries are compiled and located in `<repository>/SimulationLib`. If this is not the case, the simulation library path should be specified via the `[library path]` option.

¹ Vivado does not allow to run different simulations with different simulators automatically. If you want to do that locally, every time you want to switch from one simulator to another, you have to go to the simulation settings. The setting will be valid at project level, so for all the simulations sets. If you change the simulator by clicking in Vivado GUI, the change will not be propagated to the repository.

Parameters/Generics

Just before the synthesis starts, the Hog pre-synthesis script feeds a set of value to the design.

This is done to link the binary file with the state of the repository at the moment of synthesis. In order to do this Hog exploits VHDL generics or Verilog parameters¹. In this section the details of these generic/parameters are explained.

The values of these generics/parameters should be connected to dedicated registers that can be accessed at run time on the device (e.g. IPBus registers).

To access the Hog generics/parameters you must define the following in your top level entity:

Gener-ics/parameter name	Generics type (VHDL only)	Gener-ics/parameters size	Generics/parameters description
GLOBAL_DATE	std_logic_vector	32 bit	Last commit date. Format: ddmmyyyy (hex with decimal digits, no digit greater than 9 is used)
GLOBAL_TIME	std_logic_vector	32 bit	Last commit time. Format: 00HHMMSS (hex with decimal digits, no digit greater than 9 is used)
GLOBAL_VERSION	std_logic_vector	32 bit	Repository version. The version of the form m.M.p is encoded in hexadecimal as MMmmpppp
GLOBAL_SHA	std_logic_vector	32 bit	Repository git commit hash (SHA).
TOP_VERSION	std_logic_vector	32 bit	Top project folder version. The version of the form m.M.p is encoded in hexadecimal as MMmmpppp
TOP_SHA	std_logic_vector	32 bit	Top project folder git commit hash (SHA).
HOG_VERSION	std_logic_vector	32 bit	Hog submodule version. The version of the form m.M.p is encoded in hexadecimal as MMmmpppp
HOG_SHA	std_logic_vector	32 bit	Hog submodule git commit hash (SHA).
XML_VERSION	std_logic_vector	32 bit	(optional) IPbus xml version. The version of the form m.M.p is encoded in hexadecimal as MMmmpppp
XML_SHA	std_logic_vector	32 bit	(optional) IPbus xml git commit hash (SHA).
<MYLIB>_VERSION	std_logic_vector	32 bit	(one per library, i.e. .src list file) Version of the files contained in the .src file. The version of the form m.M.p is encoded in hexadecimal as MMmmpppp
<MYLIB>_SHA	std_logic_vector	32 bit	(one per library, i.e. .src list file) Git commit hash of the files contained in the .src file (SHA).
<MY-SUBMODULE>_SHA	std_logic_vector	32 bit	(one per submodule) Submodule git commit hash (SHA).
<MYEXTLIB>_SHA	std_logic_vector	32 bit	(one per external library) Git commit hash (SHA) of the .ext file.
FLAVOUR	integer		(integer) flavour used for generating this bit file, set if your project uses Hog flavours to produce bit files for different devices

The firmware date and time are encoded to be readable in hexadecimal so 0xA, 0xB, 0xC, 0xD 0xE, and 0xF are not used. For example the date 5 July 1952 is encoded as 0x05071952 and the time 12.34.56 is encoded as 0x00123456. To guarantee synthesis reproducibility, Hog uses the last-commit date and time rather than the synthesis date and time.

Names ending with _SHA are used for the 7-digit SHA of the git commit, being the SHA an hexadecimal number there is no ambiguity in its conversion. Names ending with _VER are used for a numeric version of the form M.m.p encoded in hexadecimal as 0xMMmmpppp. So for example version 7.10.255 becomes 0x070A00FF.

The version and hash of a subset of files is calculated using `git log` and means the latest commit (and the latest version tag) where at least one of the files was changed.

¹ Generics are used in VHDL language, parameters are used in Verilog, SystemVerilog languages.

It is worth noticing that there is no one-to-one correspondence between tag and hash, because not all the commits are tagged, so a tag can correspond to several hashes, all the ones that occurred between that tag and the previous one.

Hog will provide all the parameters/generics described in the table above, but if you do not plan to use them you can just leave them unconnected or do not add them to your top module at all. The HDL synthesiser will ignore them, maybe giving a warning.

Creating, building and simulating projects

Hog provides a series of bash scripts to create, build and simulate your projects. Of course, you are not obliged to use them and you are free to use the Vivado/Quartus GUI or Tcl console instead.

Create project

This section assumes that all the Hog list files and the project `.tcl` file have been configured as described in the previous sections.

The project can be created using shell or Vivado/Quartus Tcl console

Using shell

Open your bash shell, go to your project path and type:

```
./Hog/CreateProject.sh <project_name>
```

If you don't know the name of your project, simply issue the command without any argument and it will return the list of projects that can be created.

```
./Hog/CreateProject.sh
```

Using Vivado/Quartus Tcl console

You can also source your project `.tcl` script directly from the Vivado/Quartus Tcl console, by issuing this command:

```
source Tcl/<project_name>/<project_name>.tcl
```

Synthesis and Implement the IPs

IP synthesis can be run using shell, Vivado/Quartus GUI or Vivado/Quartus Tcl console.

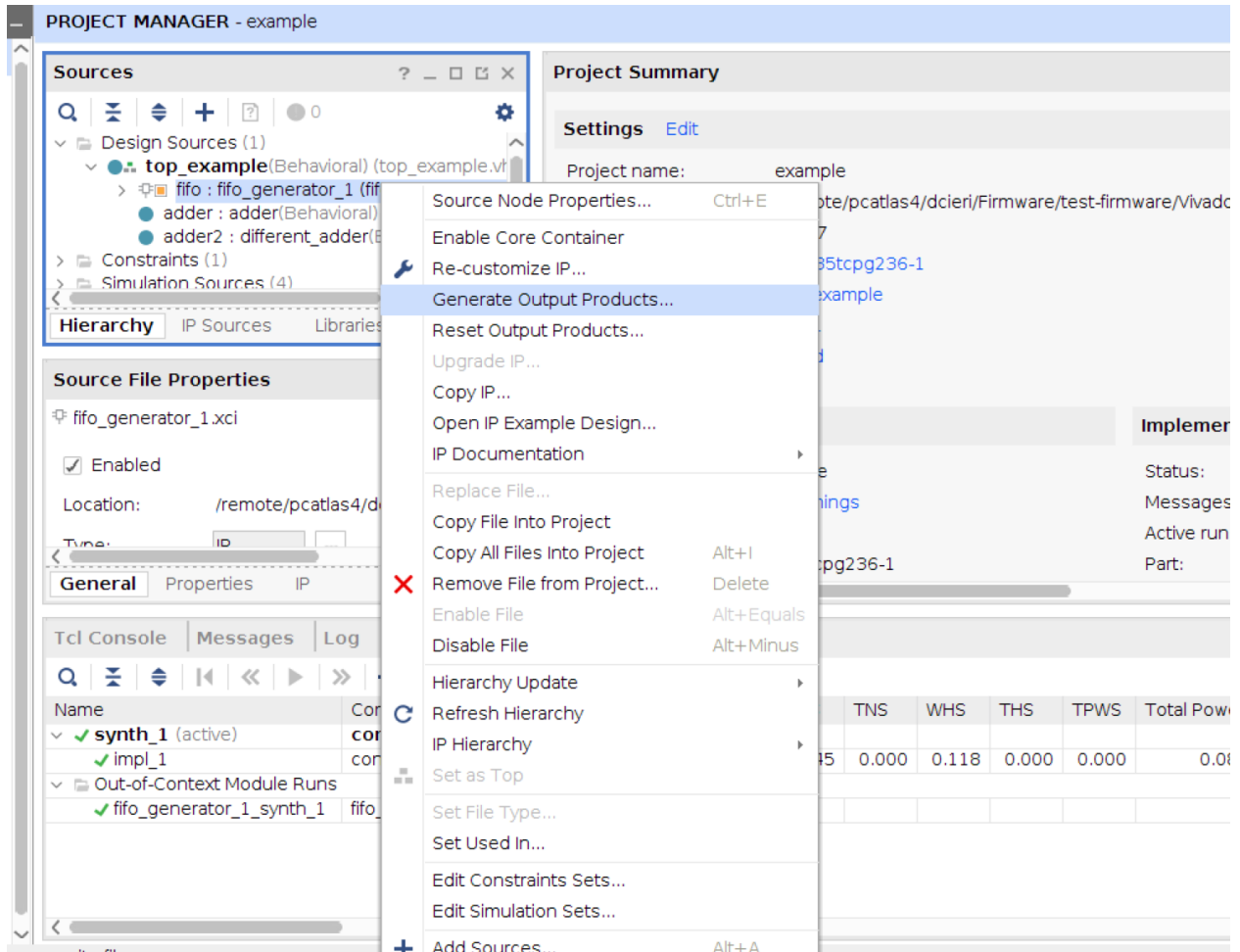
Using shell (Vivado Only)

Open your bash shell and type:

```
./Hog/LaunchIPSynth.sh <project_name>
```

Using Vivado/Quartus GUI

Right click on each IP and click the “Generate Output Products” button.



Using the Tcl console (Vivado Only)

Open Vivado Tcl console and type:

```
source ./Hog/Tcl/launchers/launch_ip_synth.tcl [-NJOBS <number of jobs>] <project_
↵name>
```

The option `-NJOBS` sets the number of jobs used to run synthesis. The default value is 4.

Synthesise your project

Project synthesis can be run using shell, Vivado/Quartus GUI or Vivado/Quartus Tcl console.

Using shell (Vivado Only)

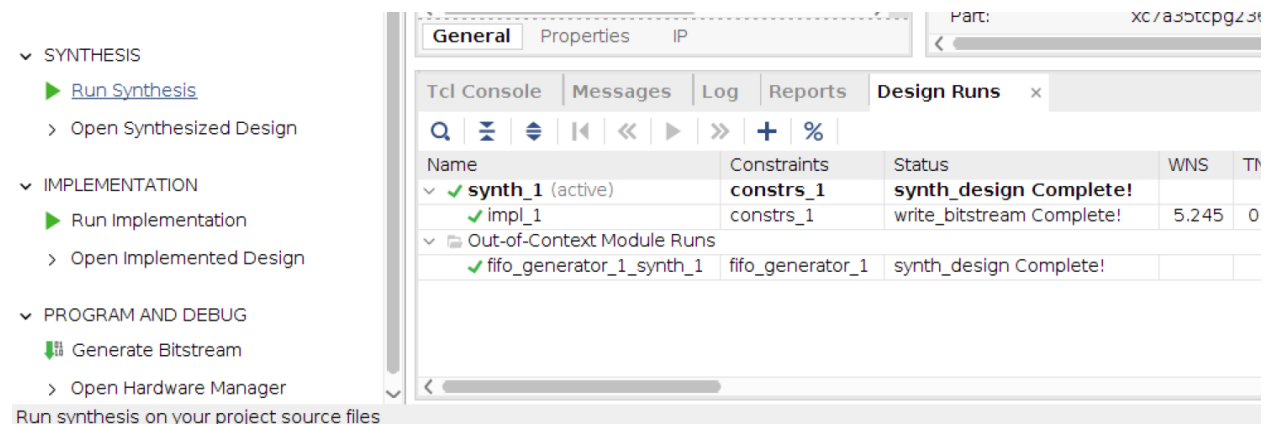
Open your bash shell and type:

```
./Hog/LaunchSynthesis.sh <project_name> [-NJOBS <number of jobs>]
```

The option `-NJOBS` sets the number of jobs used to run synthesis. The default value is 4.

Using Vivado/Quartus GUI

Click on “Run Synthesis” button (on the left).



Using the Tcl console (Vivado Only)

Open the Vivado Tcl console and type:

```
source ./Hog/Tcl/launchers/launch_synthesis.tcl [-NJOBS <number of jobs>] <project_
↪name>
```

The option `-NJOBS` sets the number of jobs used to run synthesis. The default value is 4.

Implement your project

Project implementation/bitfile generation can be run using shell, Vivado/Quartus GUI or Vivado/Quartus Tcl console.

Using shell (Vivado Only)

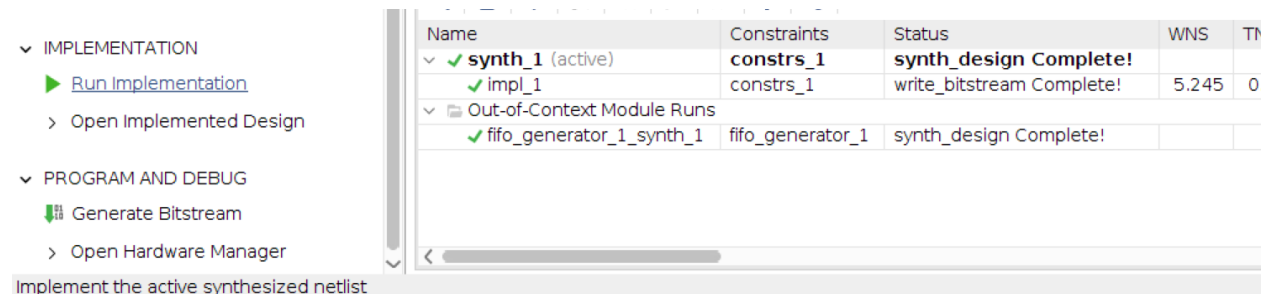
Open your bash shell and type:

```
./Hog/LaunchImplementation.sh <project_name> [-NJOBS <number of jobs>] [-no_  
↪bitstream]
```

The option *-NJOBS* sets the number of jobs used to run implementation. The default value is 4. The option *-no_bitstream* is used to skip the bitstream generation phase.

Using Vivado/Quartus GUI

Click on “Run Implementation” and “Generate Bitstream” buttons (on the left).



Using the Tcl console (Vivado Only)

Open the Vivado Tcl console and type:

```
source ./Hog/Tcl/launchers/launch_implementation.tcl [-NJOBS <number of jobs>] [-no_  
↪bitstream] <project_name>
```

The option *-NJOBS* sets the number of jobs used to run implementation. The default value is 4. The option *-no_bitstream* is used to skip the bitstream generation phase.

Run simulation

Project simulation can be run using shell, Vivado/Quartus GUI or Vivado/Quartus Tcl console. Hog supports Vivado simulator (xsim), ModelSim and QuestaSim. The simulation files and properties, such as the selected simulator, eventual wavefiles or do files are set as explained in the section [Simulation list files](#). If ModelSim or QuestaSim are used, the Vivado libraries must be compiled by the user in a directory. ModelSim/Questasim libraries can be compiled by using shell command *Hog/Init.sh* or by using the tcl commands *Hog/Tcl/utils/compile_modelsimlib.tcl* or *Hog/Tcl/utils/compile_questalib.tcl*.

If this command is used, the simulation libraries will be stored into *./SimulationLib*.

Using shell (Vivado only)

Open your bash shell and type:

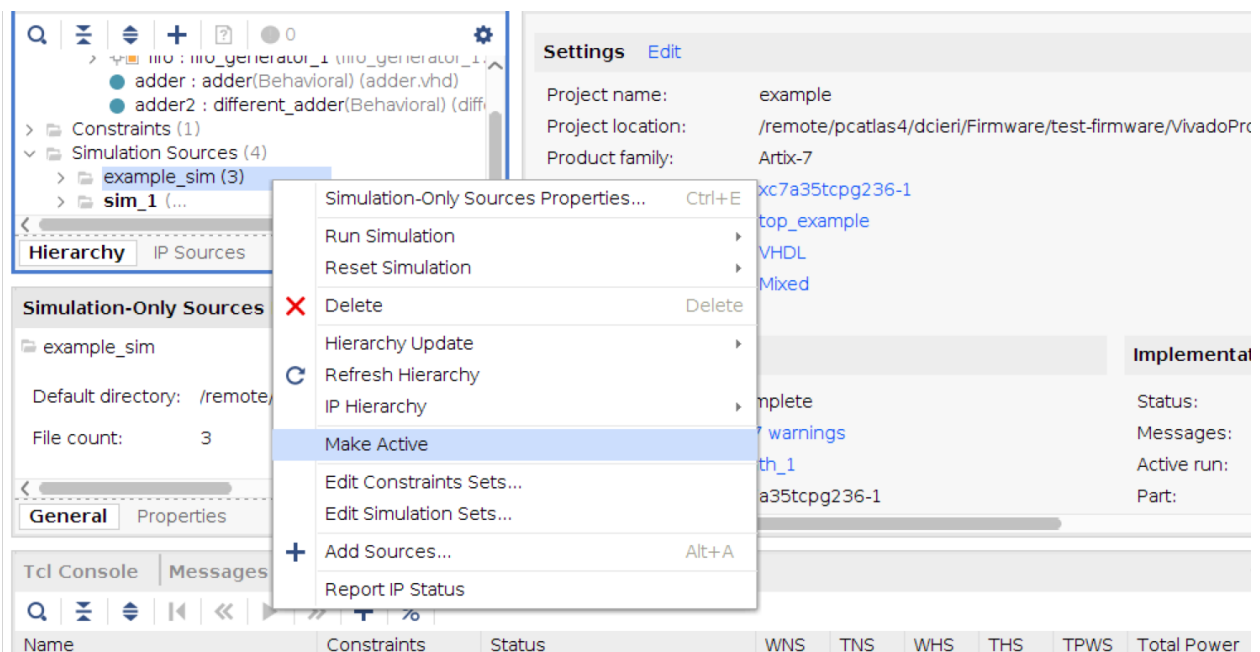
```
./Hog/LaunchSimulation.sh <project_name> [library path]
```

The option *[library path]* is the path of the compiled simulation libraries. Default: SimulationLib.

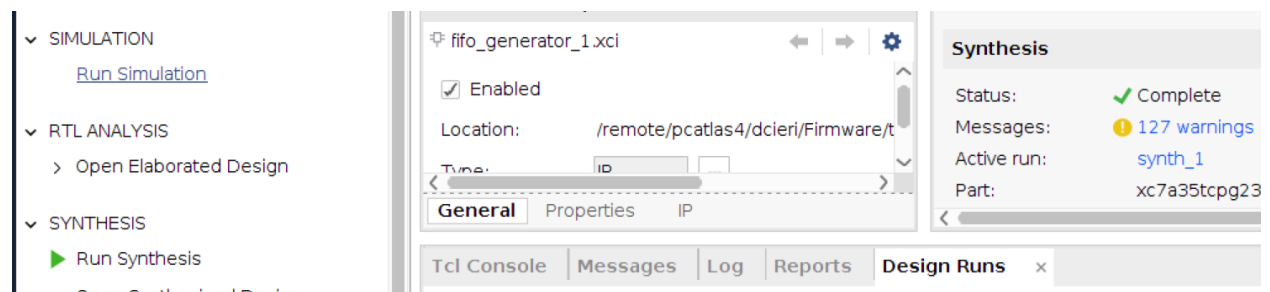
This command will launch the simulation for each `.sim` list file in your project with the chosen simulator(s).

Using Vivado/Quartus GUI

Using the GUI, you can run only one simulation set at the time. First of all select the simulation set you want to run, by right clicking on simulation set name in project sources window.



Then click on “Run Simulation” button (on the left). Note that using the GUI, Vivado or Quartus will use the simulation software specified in your *project .tcl* file



Using the Tcl console (Vivado only)

Open the Vivado Tcl console and type:

```
source ../Hog/Tcl/launchers/launch_implementation.tcl [-lib_path <library path>]
↪<project_name>
```

The option `-lib_path` is the path of the compiled simulation libraries. Default: `SimulationLib`.

This command will launch the simulation for each `.sim` list file in your project with the chosen simulator(s).

Templates

Hog provides templates for the most important file needed in your HDL repository. The templates are located in the [Hog/Templates](#) directory. Here is a list of the available templates:

top.tcl

`top.tcl` is a Tcl script to generate the HDL project, the so called “Tcl project-file”. Contains an example of all the variables used by Hog to generate a project

To use this file, copy it to the `Top/<project>/` directory, rename it to `<project>.tcl` and modify it to accommodate your needs as explained [here](#).

top.vhd

`top.vhd` is an example of top level file in VHDL. It contains the definition of the generics set by Hog to keep track of the firmware versions, as explained [here](#).

To use this file, copy it anywhere in your project, rename it and modify it to accommodate your project. Remember that the contained entity must be called `top_<project_name>` and that this file name and path must be included in a `.src` list file in `Top/project/list/` directory.

top.v

`top.v` is an example of top level file in Verilog. See `top.vhd` for details.

gitlab-ci.yml

`gitlab-ci.yml` is an example of YAML configuration file for the gitlab CI;. it contains definitions of the jobs to be run for a project in your repository. To use this file, copy it to the root folder of your repository and rename it to `.gitlab-ci.yml`. Modify the file replacing the place holder with the name of your project. If you have several projects in your repository, you should copy and paste the file content several times and change the place holder with the names of all your projects.

gitignore

`gitignore` is an example of `gitignore` file. This file tells git which files should be ignored. Files covered by a rule in this file will not be uploaded to your repository. The template contains rules for all the files used by the supported HDL compilers that should not be tracked by git. To use this file copy it to the root folder of your repository and rename it `.gitignore`. This file has special rules for the `IP` and `BD` folders. In the `IP` folder only the `.xci` files contained in a subfolder should be considered, while in the `BD` folder only the `.bd` files. This is done because the Gitlab CI needs to export the so-called artifacts so Hog has to know a priori where all the additional files will be created by the HDL synthesiser.

The provided template only works for a simple directory structure, but more complex structures can be used in Hog, as long as **all the `.xci` files** are contained in the `IP` folder and **all the `.bd` files** are contained in the `BD` folder, you can use as many subdirectories as you need. In this case you will have to write your own `.gitignore` file, remember that it is possible to use multiple `gitignore` files placed in sub directory in your repository.

doxygen.conf

`doxygen.conf` is an example of Doxygen configuration file optimised for VHDL. To use this file, copy it to a folder named `doxygen` in the root folder of your repository, modify to accommodate your project.

Hog flavour

Hog's flavour, allows the same top entity to be used in multiple projects. To differentiate the projects, an integer parameter/generic called `FLAVOUR` is provided to the top module.

What is Hog's flavour?

We have said that in a Hog-handled repository, the name of the top entity of is `top_<project name>` for each project. Well, this is not true if the project name has an extension.

So, if the project name is of the form `<name>.<ext>` and the `<ext>` is an integer non-negative number (e.g. `project.1`):

- The top entity name is `top_<name>`
- The extension is passed as a parameter/generic called `FLAVOUR` to the design.

Let's say, for example, that we have a project called `project.1` in our repository `Repo`. This means, that the project's Top directory is `Repo/Top/project.1` and the *project tcl file* is `Repo/Top/project.1/project.1.tcl`.

In this case the name of the top entity of the project will be called `project` and the number will be passed to the design in the `FLAVOUR` *parameter/generic*.

Possible use cases

This feature can be used to handle multiple FPGAs on the same board.

Different logic can be instantiated using, in VHDL, an `if ... generate` statement on the FLAVOUR value.

For example:

```
U2 : if FLAVOUR = 1 or FLAVOUR = 2 generate
    signal1 <= signal2;

Module1 : entity lib1.module1
    port map (
        CLK => clk,

        IN_Data => data_merge,
        IN_Sync => sorted_eg_Start,

        --IPBus connection
        ipb_clk    => ipb_clk,
        ipb_rst    => rst_ipb,
        ipb_in     => ipbw(N_SLV),
        ipb_out    => ipbr(N_SLV),
        Sync      => sync,
        Valid     => valid,
        Data_out   => data
    );
end generate U1;
```

Moreover subprograms can be used to achieve even more refined results on the basis of the flavour value. For example, you could have different MGTs enabled in different FPGAs and switch them on or off depending on the flavour.

So you could define a `std_logi_vector` with a function:

```
function F_MGT_QUAD_ENABLE (FLAVOUR : in integer) return std_logic_vector is
    variable V_MGT_QUAD_ENABLE : std_logic_vector(19 downto 0);
begin
    if FLAVOUR = 1 then
        V_MGT_QUAD_ENABLE := "00001111111111111111";
    elsif FLAVOUR = 2 then
        V_MGT_QUAD_ENABLE := "11101100011111111111";
    elsif FLAVOUR = 3 then
        V_MGT_QUAD_ENABLE := "111110111111111100011";
    elsif FLAVOUR = 4 then
        V_MGT_QUAD_ENABLE := "111101111111111100011";
    else
        V_MGT_QUAD_ENABLE := x"00000";
    end if;
    return V_MGT_QUAD_ENABLE;
end function F_MGT_QUAD_ENABLE;
```

And then use the string in a for loop to enable/disable an entity:

```
QUAD_ENABLE <= F_MGT_QUAD_ENABLE (FLAVOUR);

MGT_GEN : for i in 0 to num_quad_tx_rx-1
    generate
        mgt_1quad_Rx_Tx : entity work.mgt_selection_wrapper
```

(continues on next page)

(continued from previous page)

```
generic map (ENABLED      => QUAD_ENABLE(i))
port map (
    ...
);
end generate MGT_GEN;
```

Flavour can be used to minimise code repetition, even in the case that the devices have different pinout. In fact, being the project decoupled, different list files are used so xdc can be completely different.

On the other hand, if the different flavoured project share many source files (even all of them), *recursive Hog list files* can be exploited.

In fact, a list file can include another list file, so both projects could include the same list file stored everywhere in the repository. Alternatively, a common list file can be used containing the source files that are shared among the different projects, while files belonging to one specific project are listed elsewhere.

Thanks to Hog flavour, with a little bit of extra work, you can reduce the repetition of your code virtually to 0.

Additional Tcl Scripts

Hog provides a series of Tcl scripts, which execute different common tasks. These scripts are located in *./Hog/Tcl/utils*. To execute the scripts, you need to open first the Vivado Tcl Shell.

```
vivado -mode tcl
```

Run each script with the *-h* option to see the full list of arguments/options and usage example.

check-syntax.tcl

This script checks the code syntax of a Vivado project.

Arguments:

- *<project_name>*: the project name

Usage:

```
source Hog/Tcl/utils/check-syntax.tcl -tclargs <project_name>
```

compile_modelsimlib.tcl

This script compiles the ModelSim libraries needed to simulate Vivado projects with ModelSim. The libraries are stored into the directory *./SimulationLib*. Usage:

```
source Hog/Tcl/utils/compile_modelsimlib.tcl
```

compile_questalib.tcl

This script compiles the QuestaSim libraries needed to simulate Vivado projects with QuestaSim. The libraries are stored into the directory *./SimulationLib*. Usage:

```
source Hog/Tcl/utils/compile_questalib.tcl
```

get_ips.tcl

To speed-up IPs re-generation, Hog allows the user to store compiled IPs into an EOS directory and retrieve them instead of recompile them. This is particularly useful for the CI or if the project repository has been freshly cloned. The IPs are stored to EOS together with their SHA, so they are retrieved only if the `.xci` was not modified. The instructions to store the IPs to EOS are detailed in the section *IP synthesis*. `get_ips.tcl` is used to retrieve IPs from EOS. To execute this command you need to have [EOS software](#) installed on your machine.

Arguments:

- `<project_name>`: the project name

Options:

- `-eos_ip_path <IP_PATH>`: the EOS path where IPs are stored

Usage:

```
source Hog/Tcl/utils/get-ips.tcl -tclargs [-eos_ip_path <IP_PATH>] <project_name>
```

make_doxygen.tcl

This script is used to create the doxygen documentation. The doxygen configuration file must be stored into *./doxygen/doxygen.conf*. If there is no such file, the command will use *./Hog/Templates/doxygen.conf* as doxygen configuration file. You require a version of Doxygen newer than 1.8.13 installed on your machine, to execute this script

Usage:

```
source Hog/Tcl/utils/make_doxygen.tcl
```

This script is used by *Hog CI*

check_yaml_ref.tcl

This script checks that the Hog submodule SHA matches the ref in your `.gitlab-ci.yml` file. The `.gitlab-ci.yml` file defines what stages of the Hog Continuous Integration will be run. For more information, please consult the *Hog-CI chapter*.

If the two SHAs do not match, the script returns an Error, suggests few solutions to fix the problem.

This script is run by default in the pre-synthesis stage.

Usage:

```
source Hog/Tcl/utils/check_yaml_ref.tcl
```

copy_xml.tcl

This script copies IPbus XML files (see [IPbus section](#)) listed in a Hog list file and replaces the version and SHA place holders, if they are present in any of the XML files.

Arguments:

- `<xml_list_file>`: the IPbus XML list file
- `<dest_dir>`: the destination directory

Options:

- `-generate`: if set, the VHDL address files will be generated and replaced if already existing

Usage:

```
copy_xml <xml_list_file> <dest_dir> [-generate]
```

reformat.tcl

This script formats tcl scripts indentation.

Arguments:

- `<tcl_script>`: the tcl script to format

Options:

- `-tab_width <pad width>`: the tab width to be used to indent the code (default = 2)

Usage:

```
source Hog/Tcl/utils/reformat.tcl -tclargs [-tab_width <pad_width>] <tcl_script>
```

Hog and IPbus

Hog supports [IPbus](#) by handling IPbus xml files and VHDL address maps.

To use IPbus with Hog, include it as a submodule in your HDL repository (in the root folder). Include ipbus files in a `.sub` [list file](#).

Embedding of version and SHA in the xml files

Hog keeps track of the version and SHA of the xml files by means of two dedicated generics/parameters (XML_VER and XML_SHA) and dedicated node tags in the xmls.

To allow for this, your top project-directory must include an `xml` directory containing a file named: `<repository>/Top/<project_name>/xml/xml.lst`.

Note that the xml and VHDL files can be located anywhere in your project.

xml.lst

This file contains a list of the xml files used to generate the IPbus modules together with the generated VHDL address decode files, each line has the form:

```
<path_to_xml>/<address_table>.xml <path_to_vhd>/<generated_file>.vhd
```

During Pre-synthesis, Hog will loop over the files contained in this file to retrieve the SHA of the latest commit in which at least one of them was modified. The path to the generated module is needed since Hog uses an IPbus python script to verify that the generated modules correspond to the xml files.

Note that the .vhd files here are not included in your project. Typically, if you are using IPbus, you need to include those files in your project, to do that simply add them to a .src file.

IPbus xml files

Hog can back annotate the included xmls with the SHA evaluated as described above. This can be used by software to correctly assess if the used xmls correspond to the firmware loaded on the device.

You can achieve this by defining a dedicated register where to store the value of the *generic*: XML_SHA provided by Hog.

The node corresponding to this register is expected to have the following structure:

```
<node id="gitSHA" permission="r" address="0x-" tags="xmlgitsha=__GIT_SHA__"
↳description="XML git commit 7-digit SHA of top file">
```

During Pre-synthesis, Hog will replace __GIT_SHA__ with the SHA of the latest commit in which at least one of the xmls was modified. Hog will also set the XML_SHA generic in your top level to correspond to the same SHA. The user can now verify it is using the correct version of the xmls by comparing the gitSHA register content with the gitSHA register tag.

The same procedure is done for the xml version. In this case the node is expected to have the following structure:

```
<node id="Version" permission="r" address="0x-" tags="xmlversion=__VERSION__"
↳description="version of XML files">
  <node id="Patch" mask="0xffff" description="Patch Number"/>
  <node id="Minor_Version" mask="0xff0000" description="Minor Version Number"/>
  <node id="Major_Version" mask="0xff000000" description="Major Version Number"/>
</node>
```

The __VERSION__ will be set to the version of the xml files taken from the last tag in which at least one of the xml files included in xml.lst was modified. The same value will be reported in the XML_VER generic of the top level of your project.

Check address maps against xml file

Hog provides a script Hog/Tcl/Utils/copy_xml.tcl, it can be run from Vivado shell or using tclsh (provided that you installed the tcllib package¹), with this syntax:

```
Hog/Tcl/Utils/copy_xml.tcl <XML list file> <destination directory> [-generate]
```

¹ To install the tcllib package on a Linux distribution you can use `sudo yum install tcllib` or `sudo apt-get install tcllib` depending on your distribution.

This script will copy all the IPbus xml files listed in `<XML list file>` in the `<destination directory>` creating it if necessary.

Moreover it will perform the substitution of the `__GIT_SHA__` and `__GIT_VERSION__` placeholders as described above. This is useful as ipbus xml files need to be all in the same directory to work.

If the `gen_ipbus_addr_decode` IPbus script file is in `yut PATH` and works correctly, this script will also verify each xml file against its VHDL address map to see if they match. It will do this ignoring blank lines and comments and will give warnings if it find some mismatch.

Generation of VHDL address maps

If the `-generate` options is used, this script will use the `gen_ipbus_addr_decode` to generate the VHDL address map files and replace them if necessary.

You can run this script with the `-generate` option (even from the Vivado Tcl console) after you have modified the xml to regenerate the VHDL files automatically.

This script (without the `-generate` option) is also run automatically in the pre-synthesis script, and copies your xml in to `<repository>/bin/<git describe>/xml`. If the `gen_ipbus_addr_decode` is available and working, the verification is done as well.

List of supported tools

Below you can find a list of supported tools.

The Hog team does NOT provide access to the listed tools and it does NOT grants any help on the usage of the tools.

The list is intended as a list of tools you can use in an Hog compatible project and be sure The Hog style CI will be able to run on your project. The following list is not exclusive, meaning other tools might be used in your project and still work with the Hog CI scripts.

- Vivado (earliest tested: `<2017.1>` latest tested: `<2019.2>`)
 - QuestaSim¹ (earliest tested: `<10.7a>` latest tested: `<2019.2>`)
 - doxygen (earliest tested: `<1.8.17>` latest tested: `<1.8.17>`)
 - git (earliest tested: `<2.7.0>` latest tested: `<2.26.2>`)
-

Hog git hooks

Hog supports a series of special git hooks serving different purposes. The full list of the special keywords is described in this section.

¹ Depends on Vivado version used. Check the compatibility list in: <https://www.xilinx.com/support/answers/68324.html>

Merge Request description keywords

When a Merge Request is created, the *description* field can be filled with the following keywords:

- **MAJOR_VERSION**: indicates that a new major version will be released. After the branch is merged, the new tag will be `v<M+1>..` (where `M` = major, `m` = minor and `p` = patch).
- **MINOR_VERSION**: indicates that a new minor version will be released. After the branch is merged, the new tag will be `v.<m+1>.` (where `M` = major, `m` = minor and `p` = patch).

Note: If neither of the two flags is selected, by default the new tag will be `v.<p+1>` (where `M` = major, `m` = minor and `p` = patch).

Commit keywords

- **FEATURE**: if a commit message contains the **FEATURE** keywords plus a description message, the message will be automatically added to the git changelog after the branch has been merged.

1.7.6 Hog Continuous Integration

Setting up Hog Continuous Integration

Hog Continuous Integration makes use of the [Gitlab CI/CD tool](#). Both the Gitlab repository and your local area must be set-up to work properly with Hog CI. In this paragraph, we assume that we are working with a Gitlab Project called `MyProject` under the Gitlab group `MyGroup`. Please, replace these with the actual names of your project and group.

Preliminary requirements

To run the Hog-CI, you need a CERN service account. If you don't have one, you can easily request it [here](#). The service account will run the Hog CI. For that, it needs to have access to your local repository.

- Go to `https://gitlab.cern.ch/MyGroup/MyProject/-/project_members` and give *Maintainer* rights to your service account
- Log in to Gitlab with your service account and create a private access token with API rights [here](#)

Once you have your service account, you should also get 1 TB of space on EOS, that can be used to store the results of Hog CI. If, for some reasons, your service account doesn't have space on EOS, you could request it [here](#).

Set up your personal Gitlab CI YAML

Gitlab CI uses a [YAML configuration file](#) to define which commands it must run. By default this file is called `.gitlab-ci.yml` and must be stored in the root folder of your repository. Hog cannot provide a full YAML file for your project, but a template file can be found under `Hog -> Templates -> gitlab-ci.yml` as a reference. For example, suppose we want to write the `.gitlab-ci.yml` configuration file to run the Hog project `my_project` on the CI. This file will actually include the Hog `hog.yml` configuration file, where the CI stages are defined. To include the reference to the Hog parent file, add at the beginning of your `.gitlab-ci.yml`

```
include:
- project: 'hog/Hog'
  file: '/hog.yml'
  ref: 'vX.Y.Z'
```

Here you must substitute 'vX.Y.Z' with the version of Hog you want to use. The version of Hog **MUST** be specified. If you fail to do so, the CI will pick up the parent configuration file from the latest Hog master branch. This is discouraged, since Hog development could lead to not back-compatible changes that could break your CI. Moreover the pre synthesis script will check that the reference in your `.gitlab-ci.yml` file is consistent with your local Hog submodule, giving a Critical Warning if the two don't match.

Now, you need to define the stages you want to run in the CI for our project. Hog CI runs always the stages that are not project-specific (e.g. *Merge*), therefore there is no need to declare them in your file. To add a stage `stage_1` for your `my_project`, use the following syntax:

```
stage_1:my_project:
  extends: .stage_1
  variables:
    extends: .vars
    VARIABLE: <variable_value>
```

In this snippet the first line is the stage name, i.e. you are defining a stage named 'stage_1:my_project'. The second line tells the script that the stage is an extension of '.stage_1' defined in the parent `hog.yml` file. The third line starts the variable declaration section of the script. Since your script extends `.stage_1`, then it must define the variable used by this script. The line `extends: .vars` informs the variables section that it is an extension of the `.vars` object defined in `hog.yml`. The last line shows how to set the value for one named `VARIABLE` defined in the `.vars` object.

So, for example, if you want to add a *Creation* stage for your `my_project`, you should add to the `.gitlab-ci.yml`, the following lines:

```
create_project:my_project:
  extends: .create_project
  variables:
    extends: .vars
    PROJECT_NAME: my_project
```

A more detailed description of the CI stages and their YAML configuration can be found [here](#)

Remove merge commit

- Go to <https://gitlab.cern.ch/MyGroup/MyProject/edit>
- Expand **Merge Request settings**
- Select *Fast-forward merge*

Merge requests

Collapse

Choose your merge method, merge options, merge checks, merge suggestions, and set up a default description template for merge requests.

Merge method

This will dictate the commit history when you merge a merge request

- ☐ Merge commit
Every merge creates a merge commit
- ☐ Merge commit with semi-linear history
Every merge creates a merge commit
Fast-forward merges only
When conflicts arise the user is given the option to rebase
- ☒ Fast-forward merge
No merge commits are created
Fast-forward merges only
When conflicts arise the user is given the option to rebase

Pipeline configuration

- Go to https://gitlab.cern.ch/MyGroup/MyProject/-/settings/ci_cd
- Expand *General pipelines*
- Select *git clone*
- Set *Git shallow clone* to 0
- Set *Timeout* to a long threshold, for example 1d

General pipelines

Collapse

Customize your pipeline configuration, view your pipeline status and coverage report.

Git strategy for pipelines

Choose between `clone` or `fetch` to get the recent application code ?

- ☒ **git clone**
Slower but makes sure the project workspace is pristine as it clones the repository from scratch for every job
- ☐ **git fetch**
Faster as it re-uses the project workspace (falling back to clone if it doesn't exist)

Git shallow clone

The number of changes to be fetched from GitLab when cloning a repository. This can speed up Pipelines execution. Keep empty or set to 0 to disable shallow clone by default and make GitLab CI fetch all branches and tags each time.

Timeout

If any job surpasses this timeout threshold, it will be marked as failed. Human readable time input language is accepted like "1 hour". Values without specification represent seconds. ?

Set-up Runners

Unfortunately, we cannot use shared runners as the necessary software (Xilinx Vivado, Mentor Graphics Questasim, etc.) are not available. The download, installation and licensing processes would have to be done at each time that the CI is started, slowing down the entire process.. As a consequence, you need to set-up your own physical or virtual machines. Please refer to [Setting up a Virtual Machines](#) section for more information.

Now take the following actions:

- Go to Settings -> CI/CD
- Expand Runners
- On the right click `Disable shared runners for this project`
- On the left enable the private runners that you have installed on your machines.

Shared Runners

GitLab Shared Runners execute code of different projects on the same Runner unless you configure GitLab Runner Autoscale with MaxBuilds 1 (which it is on GitLab.com).

Disable shared Runners for this project

Environment variables

- Go to Settings -> CI/CD
- Expand Variables

The following variables are **needed** for Hog-CI to work, so if any of them is not defined, or defined to a wrong value, Hog-CI will fail.

Name	Value
HOG_USER	Your service account name (e.g. my_service_account)
HOG_EMAIL	Your service account's email address (e.g. service_account_mail@cern.ch)
HOG_PASSWORD	The password of your service account (should be masked)
HOG_PATH	The PATH variable for your VM, should include Vivado bin directory
HOG_PUSH_TOKEN	The push token you generated for your service account (should be masked)
HOG_XIL_LICENSE ¹	Should contain the Xilinx license servers, separated by a comma

¹ For Quartus there will be a dedicated variable in the future

With the following **optional** variables you can configure the behaviour of Hog-CI:

Name	Value
EOS_MGM_URL	Set the EOS instance. If your EOS storage is a user storage use <code>root://eosuser.cern.ch</code> . For EOS projects, have a look here
HOG_UNOFFICIAL_BIN_EOS_PATH	The EOS path for the binary files produced by the CI
HOG_OFFICIAL_BIN_EOS_PATH	The EOS path for archiving the official binary files of your project
HOG_USE_DOXYGEN	Should be set to 1, if you want the Hog CI to create the doxygen documentation of your project
HOG_CHECK_SYNTAX	Should be set to 1, if you want the Hog CI to run check syntax
HOG_CHECK_YAMLREF	If this variable is set to 1, Hog CI will check that “ref” in <code>.gitlab-ci.yml</code> actually matches the gitlab-ci file in the Hog submodule
HOG_IP_EOS_PATH	The EOS path where to store the IP generated results. If not set, the CI will synthesise the IPs each time
HOG_NO_BITSTREAM	If this variable is set to 1, Hog-CI runs the implementation but does NOT run the write_bitstream stage
HOG_CREATE_OFFICIAL_RELEASE	If RELEASE is set to 1, Hog-CI creates an official release note using the version and timing summaries taken from the artifact of the projects.
HOG_SIMULATION_LIB_PATH	The path in your VM, where the Simulation Lib files are stored (Vivado only)
HOG_TARGET_BRANCH	Project target branch. Merge request should start from this branch. Default: master
HOG_NJOBS	Number of CPU jobs for the synthesis and implementation. Default: 4
HOG_IP_NJOBS	Number of CPU jobs for the synthesis and implementation. Default: 4

EOS space (Optional)

The Gitlab CI will produce some artefacts. These include the resulting binary files of your firmware projects and, optionally, the Doxygen documentation html files. Hog has also the ability to copy these files into a desired EOS repository. To enable this feature, we have to specify the following environmental variables: **EOS_MGM_URL**, **HOG_UNOFFICIAL_BIN_EOS_PATH**, **HOG_OFFICIAL_BIN_EOS_PATH**.

If you wish to have your files to be accessible in a web browser, you should create a web page in EOS, following [these instructions](#). For a personal project, by default, the website will be stored in `/eos/user/<initial>/<userID>/www`. The Hog EOS paths must be then sub-folders of the website root path. To expose the files in the website, follow these [instructions](#).

Gitlab Work-Flow

To fully exploit Gitlab features, the work-flow should start with the creation of a new issue and a related merge request and branch.

To do this, go to the Gitlab website and navigate to your repository.

Click on issues and open a new issue describing the fix you are to implement or the new feature you want to introduce. Once you have an issue, you can open a merge request marked as *WIP* (work in progress) and a new branch simply by clicking `Create merge request` inside the issue overview.

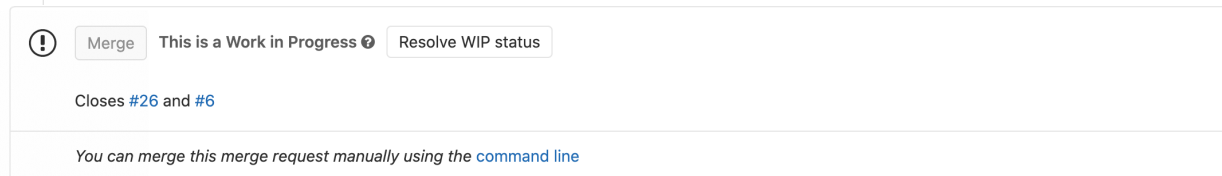
When creating the merge request, you can write the *MINOR_VERSION* or *MAJOR_VERSION* keywords in the merge request description, to tell Hog how to tag the repository once the source branch is merged into your official branch.

Now that you have a new branch connected to the issue, on your local machine, navigate to your local project folder and checkout the new branch.

You can now develop your new feature or commit the code you have.

Once you are done with your changes, you are ready to start the Hog CI.

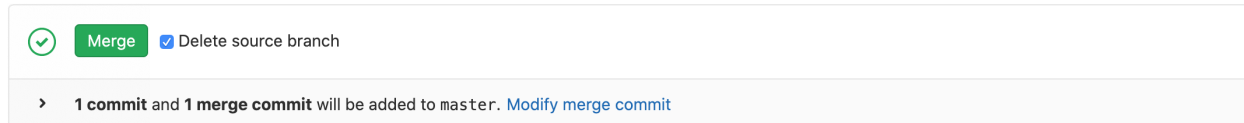
To do this, you need to solve the WIP status. This can be done either by going to the Merge Request page in the Gitlab web-site and click on the `Resolve WIP` status,



or (thanks to a Hog feature) you can start your commit message with `RESOLVE_WIP :` from the command line.

```
MyProject> git add new_files
MyProject> git commit -m "RESOLVE_WIP: <my message>"
MyProject> git push
```

If you opt for the website method, remember that you need to push another commit afterwards to start the CI. Once the pipeline passes and your changes are approved, if required, you can merge your source branch simply by clicking on the merge button in the merge request.



Create a Merge Request without an issue

You can avoid using issues by creating a new branch and a merge request connected to your branch. You can still use the nice WIP feature by adding `[WIP]` or `WIP :` at the beginning of the title of the merge request: the merge request will be [marked as work in progress](#).

Commit your code and accidental commits

If you have already some uncommitted/committed new features, you should create a new branch, commit your code there and create a new merge request when ready.

If you have already committed your changes to a wrong branch (e.g. the `master`) simply reset that branch to the latest correct commit, e.g.

```
MyProject> git reset --hard <latest_correct_commit_hash>
MyProject> git push origin master
```

Create a new branch, check it out and commit your code there. To avoid direct commit to the master (or official) branch, you can configure the repository to forbid pushing directly to the some special branch, for example master.

To do this go to Settings and then Repository on Gitlab webiste. Then expand Protected Branches and add the branches that you want to protect.

Increasing version number

Hog uses a version of the form $vM.m.p$. Where M is the major version, m is the minor version and p is the *patch*. You will be able to increase the different levels of version by editing the merge request description.

The major version number can be increased by placing `MAJOR_VERSION` in the merge request description. While merging the merge request Hog will read the description, find the `MAJOR_VERSION` keyword and increase the major version counter. This will also reset the minor and patch counters.

The minor version number can be increased by placing `MINOR_VERSION` in the merge request description. While merging the merge request Hog will read the description, find the `MINOR_VERSION` keyword and increase the minor version counter. This will also reset the patch counters.

The patch number will be increased automatically at each accepted merge request. While merging the merge request Hog will read the description, find no keyword and increase the patch counter.

Examples

Let's suppose the last tag of your firmware is v1.2.4, the possible scenarios are:

Merge request description	Original version	Final version
without any keyword	v1.2.4	v1.2.5
contains <code>MINOR_VERSION</code> keyword	v1.2.4	v1.3.0
contains <code>MAJOR_VERSION</code> keyword	v1.2.4	v2.0.0

Gitlab Release Notes

When creating a new tag, Hog CI will also create a new release. Hog has the ability to write automatically the release note, by looking at the merge request commit messages. If you want a commit message to be included in the release note, you should start your commit message with the `FEATURE:` keyword. For example:

```
git commit -m "FEATURE: Some awesome update"
```

Configure your YAML file

In this paragraph, we describe the stages of the Hog CI pipelines that need to be configured in your local `.gitlab-ci.yml` file. Only the stages that are *project-specific* must be defined in this file. General stages are directly included from the `hog.yml` reference file and are not configurable.

The configurable stages are:

- Creation
- Simulation
- IP
- Synthesis
- Implementation

Creation Stage

The Creation stage calls the `create_project` function and creates the Quartus/Vivado project, using the `Hog/CreateProject.sh` bash script.

It can be configured with the following variables:

- `PROJECT_NAME` : (**Mandatory**) name of the Hog project you want to create
- `HOG_CHECK_SYNTAX` : (**Optional**) if 1, it checks the syntax of the created project. Better if defined globally as an environmental variable in your *Gitlab repository*
- `HOG_CHECK_YAMLREF` : (**Optional**) if 1, it checks that the `hog.yml` file referenced in your `.gitlab-ci.yml` is the same as the one included in your Hog submodule. Better if defined globally as an environmental variable in your *Gitlab repository*

The resulting stage in your `.gitlab-ci.yml` file, for the project `my_project` is,

```
create_project:my_project:
  extends: .create_project
  variables:
    extends: .vars
    PROJECT_NAME: my_project
```

Simulation Stage

The Simulation stage calls the `simulate_project` function and launches a behavioural simulation for each `.sim` list file in your Hog project. By default, simulation is executed using *Modelsim*. If you wish to use another simulation software, add the following line to the top of your `.sim` list file:

```
Simulator xsim # For Vivado Simulator
Simulator questa # For QuestaSim
Simulator modelsim # For Modelsim
```

`simulate_project` requires two variables:

- `PROJECT_NAME` : (**Mandatory**) name of the Hog project you want to simulate
- `HOG_SIMULATION_LIB_PATH`: (**Mandatory for Questa/Modelsim**) Path to the compiled simulation library in your VM. It shall be defined in your *Gitlab CI/CD variables*.

The resulting stage in your `.gitlab-ci.yml` file, for the project `my_project` is,

```
simulate_project:my_project:
  extends: .simulate_project
  variables:
    extends: .vars
    PROJECT_NAME: my_project
```


IP Stage

The IP stage calls the `synthesise_ips` function and generates the synthesis and implementation products for the IPs included in your project. It is configured with the following variables.

- `PROJECT_NAME` : **(Mandatory)** name of the Hog project to open
- `HOG_IP_NJOBS` : **(Optional)** number of jobs to generate the IP products. It shall be defined in your *Gitlab CI/CD variables*. Default: 4
- `HOG_IP_EOS_PATH`: **(Optional)** path to the EOS folder where the IP generated results are stored. If defined, the stage will copy the IP products from EOS without relaunching the IP synthesis/implementation to speed up the pipeline. If the IP products on EOS are outdated, the script will regenerate and upload them to EOS. It shall be defined in your *Gitlab CI/CD variables*.

The resulting stage in your `.gitlab-ci.yml` file, for the project `my_project` is,

```
synthesise_ips:my_project:
  extends: .synthesise_ips
  variables:
    extends: .vars
    PROJECT_NAME: my_project
  dependencies:
    - create_project:my_project
```

Synthesis Stage

The Synthesis stage calls the `synthesise_project` function and synthesise your project. It is configured with the following variables.

- `PROJECT_NAME`: **(Mandatory)** name of the Hog project to open
- `HOG_NJOBS`: **(Optional)** number of jobs to run the synthesis. It shall be defined in your *Gitlab CI/CD variables*. Default: 4

The resulting stage in your `.gitlab-ci.yml` file, for the project `my_project` is,

```
synthesise_project:my_project:
  extends: .synthesise_project
  variables:
    extends: .vars
    PROJECT_NAME: my_project
  dependencies:
    - synthesise_ips:my_project
```

Implementation Stage

The Implementation stage calls the `implement_project` function and runs the implementation of your project. It is configured with the following variables.

- `PROJECT_NAME`: **(Mandatory)** name of the Hog project to open
- `HOG_NJOBS`: **(Optional)** number of jobs to run the synthesis. It shall be defined in your *Gitlab CI/CD variables*. Default: 4
- `HOG_NO_BITSTREAM`: **(Optional)** If set to 1, the script will not write a binary file for your project. Default: 0

The resulting stage in your `.gitlab-ci.yml` file, for the project `my_project` is,

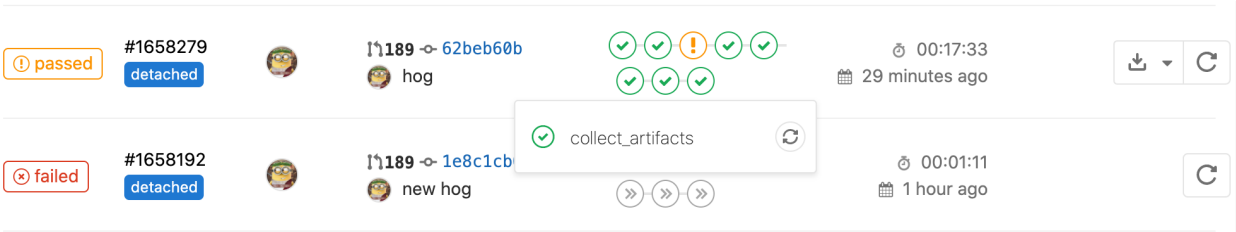
```
implement_project:my_project:
  extends: .implement_project
  variables:
    extends: .vars
    PROJECT_NAME: my_project
    HOG_NO_BITSTREAM : 1 # No bitstream will be produced
  dependencies:
    - synthesise_project:my_project
```

Hog CI Products

In this paragraph, we describe the output products of the Hog CI pipelines.

Merge Request Pipeline Products

The Merge Request pipeline generates a bin folder, where it stores the output products for each Hog project that has been run over the CI. It can be browsed, by opening the collect_artifacts stage of your pipeline and then clicking on Download or Browse on the right sidebar.



For each project, it creates a sub-folder with the following format:

```
<project_name>-<latest-tag>-<number_mr_commit>-<git-SHA>
```

For example, in our TestFirmware, we have four Hog projects: bd_design, example, proj.1 and proj.2, and the bin folder content, looks like:

Name	Size
..	
bd_design-v0.2.64-2-g62beb60	
example-v0.2.64-2-g62beb60	
proj.1-v0.2.64-2-g62beb60	
proj.2-v0.2.64-2-g62beb60	
note.md	811 Bytes

Inside each project sub-folder, you will find the bitstream files, a txt file with the timing report (timing_*.txt), a txt file with the version summary (version.txt), a report folder containing the Vivado/Quartus reports and an xml folder for possible address tables.

The Merge Request pipelines writes also notes with the resulting timing and version status in the Gitlab MR page, for faster control.

.

Doxygen documentation

If configured (`HOG_USE_DOXYGEN` set to 1), Hog CI creates also the Doxygen documentation for the entire repository. This documentation can be browsed by opening `doxygen` stage artefacts in the Gitlab web page.

To create Doxygen documentation Hog uses the `[make_doxygen.tcl script](../01-Hog-local/06-Hog-utils.md#make_doxygen)`

EOS Unofficial

If configured (`HOG_UNOFFICIAL_BIN_EOS_PATH` defined), the Hog CI will also create a folder, named as the git SHA, in `HOG_UNOFFICIAL_BIN_EOS_PATH`, where it copies the content of the `bin` folder. If the CI produced the Doxygen documentation, the html version is also copied in the same folder, inside a `Doc-*` sub-folder.

Tag Pipeline Products

The Tag pipeline creates the Gitlab Release note, as described [\[here\]\(03-gitlab-workflow.md#Gitlab Release Notes\)](#) and, if `HOG_UNOFFICIAL_BIN_EOS_PATH` and `HOG_OFFICIAL_BIN_EOS_PATH` are defined, it copies the content of the latest git SHA folder in `HOG_UNOFFICIAL_BIN_EOS_PATH` to a new folder in `HOG_OFFICIAL_BIN_EOS_PATH`, named as the new tag. If `doxygen` has been also run, the newly generated documentation is copied also in the `Doc` folder inside `HOG_OFFICIAL_BIN_EOS_PATH`.

Finally, if the copy is successful, the CI deletes all the sub-folders in `HOG_UNOFFICIAL_BIN_EOS_PATH` related to the Merge Request just merged.

Setting up a Virtual Machines for firmware implementation

To run the Hog CI, you need a dedicated Virtual Machine, since the available shared runners are not ideal to execute heavy software like Vivado or Quartus.

Setting up a dedicated Virtual Machine

In this section, you can find more information on how to set up your private Gitlab runner. Instructions are provided assuming you have access to the CERN computing resources. If this is not the case, you can still use Hog assuming that you have access to machine running CentOS 7 set up as a Gitlab runner.

In this case, you can ignore the next section and jump directly to [\[Install Gitlab runner\]\(#install_gitlab_runner\)](#)

Create a CERN Openstack Virtual Machine

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources through a data-centre, managed and provisioned through APIs with common authentication mechanisms. Openstack provides you with a [dashboard](#) from which you can manage VM instances.

More information on Openstack can be found in the [Openstack Dashboard Documentation](#).

To create a new VM, you have to connect to the [CERN Openstack dashboard](#) and create a new instance.

Openstack instances come with different flavours, meaning that you can allocate only a fixed amount of each resource to each VM.

Before creating an new instance you can add a new disk that you can use to install the needed tools. To do this go under `Volumes -> Volumes` on the left navigation bar. Once the Volumes summary appears you can click on `+ Create Volume` and follow the instructions therein. We recommend having at least a 40GB HD

Once you have obtained a custom flavour and a dedicated disk, you can create a new instance. Navigate to `Compute > Instances`, once you get to the instances summary click on `Launch Instance`. Fill in the required information in the form that will appear. Under the `Source` tab select an updated *CC7 image*: this will generate a VM running under CentOS 7. Select the custom flavour in the `Flavor` tab. Generate a new key pair and save the private key, this will be needed later to access your VM.

Once a new instance is running (note it might take a few minutes to be generated) attach the Volume you created to the VM. This can be done through the drop down menu on the right side of the instance summary by clicking on `attach volume`.

You can now connect to your machine through ssh.

NOTE your machine is not fully public yet (reference the Openstack manual for this). This means your VM will be accessible only from the CERN domain. If you are not on the CERN domain, connect to a CERN public machine (lxxplus) and then to your machine.

```
ssh -i private-key.pem <machine_ip_or_name>
```

Once you are logged into your machine, change the root password:

```
sudo password root
```

Please follow the IT recommendation when choosing a new password. Mount the volume you created, make sure you own it, format it, etc...

```
sudo su                                # become root
mkfs.ext3 /dev/<diskname>               # format the disk
mkdir /mnt/vd                          # create mounting point for the disk
mount /dev/<diskname> /mnt/vd           # mount the disk
chown -hR <username> /mnt/vd          # own the disk
```

NOTE there is no need to add this disk to `/etc/fstab` for automatic mounting, since Hog will do this automatically.

You are now ready to install your favourite tools!

Installing HDL tools

You can now install the licensed software (Xilinx Vivado, Mentor Graphics Questasim, ...) that you plan to use in your project. *NOTE* you are the one responsible for correctly licensing the software!

Install Gitlab runner

Information on how to install a new Gitlab runner on your VM can be found [here](#)

Allowing concurrent jobs on a single Openstack Virtual Machine

- Log into your VM
- Open with your preferred editor (with sudo rights) the file `/etc/gitlab-runner/config.toml`
- In the `global` section add `concurrent = NUMBER_OF_CONCURRENT_CPU`: limits how many jobs globally can be run concurrently. These changes apply to all the runners on the machine independently of the executor [docker, ssh, kubernetes etc]
- In the `runner` section add `limit = MAX_NUMBER_OF_CONCURRENT_JOB_PER_RUNNER`: Limit how many jobs can be handled concurrently by this token. Suppose that we have 2 runners registered by 2 different tokens, then their limits could be adjusted separately : runner-one limit = 3, runner-two limit =5,
- In the `runner` section add `request_concurrency = NUMBER_OF_CONCURRENT_REQUESTS_PER_NEW_JOBS` : Limit number of concurrent requests for new jobs from Gitlab (default 1)
- [Have a look here for more info](#)
- Example `config.toml`

```
concurrent = 4
check_interval = 0

[session_server]
  session_timeout = 1800

[[runners]]
  limit = 4
  request_concurrency = 4
  name = "Hog vivado runner on mypc"
  url = "https://gitlab.cern.ch"
  token = "ibsaiddasdhubavsuod"
  executor = "shell"
  [runners.custom_build_dir]
  [runners.cache]
  [runners.cache.s3]
  [runners.cache.gcs]
```

Hog set-up on the Gitlab runner

Hog will need the Virtual Machine to be properly set-up as Gitlab runner.

- clone the [Hog VM-setup repository](#) somewhere accessible from the VM, e.g. on your AFS home
- `ssh` into your virtual machine as yourself
- become root (`su -u username`)
- export the following system variables:
 - **HOG_USERNAME**= The name of your service account
 - **HOG_VIVADO_DIR**= Path of your Vivado SDK installation directory containing the `xsetup` executable (not required if you run the script with the `-x` flag)
 - **HOG_TOKEN**= A valid Gitlab private runner token: Go to Settings -> CI/CD and expand the Runners tab. The registration token in `Specific Runners` column.
 - **HOG_USERGROUP**= The name of your user group, e.g. “zp” for ATLAS
- go to the `VM-setup` directory and launch the **hog-vm-setup.sh** script
- once the script has finished, you can login to the VM as your service account

The new Gitlab runner should now appear in your Gitlab repository, in Settings -> CI/CD -> Runners -> Specific Runners. Remember to enable it for your project.

Hog Continuous Integration

Hog Continuous Integration (CI) makes use of the [Gitlab CI/CD tool](#). The main features of Hog CI are:

- Control that merging branches are up-to-date with targets
- Create and builds Vivado/Quartus projects
- Generate FPGA binary and report files with embedded git commit SHA
- Automatically generate VHDL code documentation using *doxygen*
- If configured, store IP generated files and implementation project results in a user-defined EOS folder
- Automatically tag the Gitlab repository and creates *release notes*

Three pipelines are employed, triggered by the following actions:

- **Merge Request Pipeline**: triggered by each commit into a *non-WIP* merge request branch
- **Master Pipeline**: triggered by each commit into the master branch
- **Tag Pipeline**: triggered by the creation of a new official tag (starting with “v*”)

Merge Request Pipeline

The *Merge Request* pipeline simulates, synthesises and implements the chosen HDL projects. If specified, it stores the resulting outputs to an EOS repository and creates the doxygen documentation.

The stages of the Merge Request pipeline are the following:

1. *Merge*: checks that all the required Hog environmental variables are set up and that the source branch is not outdated with respect to the target branch. If it is, the pipeline fails and asks the user to update the source branch.
2. *Creation*: creates a Vivado/Quartus project for each project specified in the `.gitlab-ci.yml` file. It checks also that the Hog submodule in the repository is the same as the one specified in the CI configuration file. Finally, for Vivado projects, it checks the syntax of the HDL codes, before moving to the next stage.
3. *IP*: generates the synthesis and implementation files for the IP in each project. An option to use the EOS repository to store the IP results and retrieve them to speed up the pipeline, can be enabled.
4. *Synthesis*: synthesises the projects.
5. *Implementation*: Implements the projects and creates the bitstreams. An option to disable the bitstream creation is available. During this stage the implementation timing results and project version are also written in the merge request page.
6. *Collect*: Collects all the artefacts from the previous stages. If EOS is used, it copies the implementation outputs to the EOS repository.
7. *Doxygen*: Creates the doxygen documentation and stores it to EOS if enabled.

Master Pipeline

The *Master* pipeline consists only of one stage (*Merge*), which tags the repository according to the Merge Request description. Assuming the latest tag was `vA.B.C`, the pipeline will

- increase A, if the MR description contains the keyword “MAJOR_VERSION”
- increase B, if the MR description contains the keyword “MINOR_VERSION”
- increase C, in the other cases

Tag Pipeline

The *Tag* pipeline consists of two stages:


1. *Copy*: If EOS is enabled, copies the *Merge Request* output files from the EOS unofficial to the EOS official storage, creating a new sub-folder with the name of the new tag. It also writes the release note for the new tag, with the timing results and the project version.
2. *Clean*: If EOS is enabled, cleans the unofficial storage of all the files of the merge request already merged.

1.7.7 Report a bug

We always want to improve Hog, for this reason your help is very important!

As a Hog user, you are very welcome to report bugs and suggest new features opening a new issue in our Gitlab repository: <https://gitlab.cern.ch/hog/Hog/issues>


When you report bugs, please do it one at the time. Before reporting, it is very important to check if the same bug has been already reported. Take a look at the list of the reported bugs [here at this link](#).

Open an issue using the label 

If your report is clear and effective, then the chances to have it fixed are higher. To help us debugging, in the [Description] section of the [Issue] try to describe in detail the problem that you encountered.

Let us know:

- which error/unexpected message you receive
- expected vs actual result
- if your bug is reproducible and how
- which Hog version you are using
- if you are working on Windows or Linux
- if you are using Vivado or Quartus and which version
- the link to your repository (or to where your code is online stored)


If you want to ask a question about Hog rather than report a bug you can use the label 

Please do not use any other labels apart from “Feature proposal”, “Question” and “Problem report” because they are meant for developers rather than users.

1.7.8 Suggest new features

When you suggest a feature, please do it one at the time.

Your feature will not be automatically added to the repository, but the Hog group will evaluate it. For this reason, it is important that you describe in detail your suggestion.

Open an issue using the label .

You can do that [at this link](#).

Let us know:

- if it's a missing feature or a feature that should change
- what will this save or fix
- to which part of Hog this is related to
- how you have been testing it (e.g. operative system, IDE software, Hog version etc.)

Please do not use any other labels apart from “Feature proposal”, “Question” and “Problem report” because they are meant for developers rather than users.

1.7.9 Developing for Hog

Become a member of the Hog community

You are very welcome to become an active Hog developer!

Get in contact with one of us (e.g. [Francesco Gonnella](#) or [Davide Cieri](#)), such that we can have a quick feedback of your background and your expertise.

Developing for Hog

As a Hog developer, if you want to contribute to Hog please follow these instructions:

1. go to [Hog on gitlab](#)
2. check in the issues list that your improvements/features are not already under development.
3. create a new issue by clicking on “New issue” button
 - use a meaningful short name
 - write a comprehensive description of the changes you plan to make
 - the issue must be assigned to you
 - use labels to indicate whether it is a new feature a bug-fix, etc
 - (OPTIONAL) use due date to indicate when you expect to conclude your work
4. open a new merge request
 - starting from your newly created issue
 - expand Merge Request drop-down
 - expand Create Merge Request
 - direct your merge request to `develop` branch
 - * branch name: `feature/<issue_short_name>`
 - click on “Create Merge Request”:
 - a new merge request is created
 - the merge request will be marked as WIP
 - a new development branch is created
5. clone the TestFirmware repository

```
cd path_to_workdir/
git clone --recursive https://gitlab.cern.ch/hog/test/TestFirmware.git
```

6. Create a new branch in the test firmware repository with the same name as the one in the main Hog repository

```
cd path_to_workdir/TestFirmware/
git checkout -b feature/<issue_short_name>
```

7. Move Hog to your branch

```
cd path_to_workdir/TestFirmware/Hog/
git checkout feature/<issue_short_name>
```

8. develop a test for your new feature *before writing your code*

- eventually modify an existing tests
- commit your tests

```
cd path_to_workdir/TestFirmware/  
git commit <test1> <test1> <...> -m "Adding tests for feature/<issue_short_name>:  
↪<brieftestdescription>"
```

9. develop the code for the new feature (IN THIS ORDER!)

- all code must be documented using doxygen comments in the code!
- commit your code

```
cd path_to_workdir/TestFirmware/Hog/  
git commit <file1> <file2> <...> -m "Working on feature/<issue_short_name>:  
↪<briefcommitdescription>"
```

10. test your code

- all your new test must succeed
- all existing test must succeed
- if the test fails, fix your code and commit it using `--amend`

```
cd path_to_workdir/TestFirmware/Hog/  
git commit --amend --no-edit
```

11. if your modification has any impact on the user/maintainer part, update the user manual accordingly

12. push your changes in the Hog repository

```
cd path_to_workdir/TestFirmware/Hog/  
git push
```

13. push your changes on the TestFirmware repository

```
cd path_to_workdir/TestFirmware/  
git push --set-upstream origin feature/<issue_short_name>
```

14. remove WIP status from your Merge Request - go to [Hog on gitlab](#) - navigate to your merge request - click on "Resolve WIP status" button

15. drop us a line at [Hog support](#)

16. check your Merge Request and address comments

Documenting the code

All the code written to implement new features or correct bugs must be documented. The main source of documentation is doxygen and comments in the code. The doxygen documentation is collected in a dedicated [website](#)

An example of how to document new functions

```
# @brief Brief description of this method  
#  
# After an empty line you can add a more detailed description.
```

(continues on next page)

(continued from previous page)

```
# You can even use many lines
#
# @param[in]      param_1      the description of parameter param_1, this_
↪parameter is an input to the function
# @param[out]     param_2      the description of parameter param_2, this_
↪parameter is an output to the function
#
# @returns        A description of the returned value
#
proc Example {param_1 param_1} {
    if {[info commands get_property] != ""} {
        # some Vivado specific comments that will not end up in the documentation
        return "Vivado"

    } elseif {[info commands quartus_command] != ""} {
        # some Vivado specific comments that will not end up in the documentation
        return "Quartus"
    } else {
        # Tcl Shell
        return "DEBUG_property_value"
    }
}
```

The resulting documentation will be a brief description in the list of available functions:

Functions

Example param_1 param_1
Brief description of this method. More...

Linked to a detailed description:

Example param_1 param_1

Brief description of this method.

After an empty line you can add a more detailed description. You can even use many lines

And you can use lists to some extent, as an example:

- this is one item
- this is another one

Parameters

[in] param_1 the description of parameter param_1, this parameter is an input to the function
[out] param_2 the description of parameter param_2, this parameter is an output to the function

Returns

A description of the returned value

Definition at line 20 of file hog.tcl.

The same comment style can be used also for bash scripts provided you use functions in your script.

Contributing to the Manual

This site uses MkDocs to render the Markdown files. The source is hosted on gitLab: [Hog](#)

To contribute to the user manual please read this section carefully. You should first clone the repository:

```
git clone https://gitlab.cern.ch/hog/hog-docs
```

As an alternative you can use the Web IDE directly from the Gitlab website. This allows you to preview the resulting page. If you want to do this locally and haven't set up your permissions for local Gitlab yet, follow the instructions [here](#). Everything you'll need to edit is inside the docs/ directory. Sections are represented by subdirectories within docs/ while the "Introduction" section comes from index.md. You can create further markdown files to add topics

to the section. Any change you make in the repository is propagated to the website, when you push your commits into the `master` branch.

Markdown

This manual is made in markdown, a simple language for formatting text. If you're not familiar, there is a handy cheatsheet [here](#). There are a few special cases for the specific flavor of markdown that gitLab uses (most notably for newline syntax) that are documented [here](#).

Continuous integration set-up

CI for this project was set up using the information in the [mkdocs](#) repository. The generated website is automatically deployed [here](#)

1.7.10 Stable Releases

Stable Release Hog2020.1

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F125%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 125
- Branch name: hotfix-version

Changelog

- (bugfix) pick the correct version

Stable Release Hog2020.1-2

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F128%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 128
- Branch name: hotfix-ipbus-xml-version

Changelog

- bugfix xml SHA and Version all zeroes in xml file

Stable Release Hog2020.1-3

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F136%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 136
- Branch name: hotfix-binfile

Changelog

- (bugfix) In case current commit is not contained in any tag, GetVer takes the last version from the last tag that is parent of the current commit, not the last tag in the repository as it used to be
- (bugfix) fix binfile was never created

Stable Release Hog2020.1-4

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F140%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 140
- Branch name: hotfix-releases

Changelog

- (bugfix) fixed bug preventing Hog from creating a Gitlab release, in case one of the IP is modified during by the CI

Stable Release Hog2020.1-5

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F142%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 142
- Branch name: hotfix-cleaning

Changelog

- (bugfix) fixed clean_unofficial script

Stable Release Hog2020.1.1

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F127%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 127
- Branch name: 93-resolvewip-broken

Changelog

- (bugfix) keyword to start pipeline and resolve wip status is now <RESOLVE_WIP:>

1.7.11 Beta Releases

Beta Release v1.0.10

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F140%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 140
- Branch name: hotfix-releases

Changelog

- (bugfix) fixed bug preventing Hog from creating a Gitlab release, in case one of the IP is modified during by the CI

Beta Release v1.0.11

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F142%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 142
- Branch name: hotfix-cleaning

Changelog

- (bugfix) fixed clean_unofficial script

Beta Release v1.0.4

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F123%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 123
- Branch name: hotfix-release3

Changelog

Beta Release v1.0.5

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F124%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 124
- Branch name: hotfix-release4

Changelog

- Release notes are automatically copied into the Official documentation

Beta Release v1.0.6

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F125%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 125
- Branch name: hotfix-version

Changelog

- (bugfix) pick the correct version

Beta Release v1.0.7

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F127%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 127
- Branch name: 93-resolvewip-broken

Changelog

- (bugfix) keyword to start pipeline and resolve wip status is now <RESOLVE_WIP:>

Beta Release v1.0.8

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F128%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 128
- Branch name: hotfix-ipbus-xml-version

Changelog

- bugfix xml SHA and Version all zeroes in xml file

Beta Release v1.0.9

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F136%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 136
- Branch name: hotfix-binfile

Changelog

- (bugfix) In case current commit is not contained in any tag, GetVer takes the last version from the last tag that is parent of the current commit, not the last tag in the repository as it used to be
- (bugfix) fix binfile was never created

Beta Release v1.1.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F131%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 131
- Branch name: 91-launchipsynth-sh-wants-to-copy-to-eos-even-when-it-shouldn-t

Changelog

- (bugfix) fixed bug in LaunchIPSynth.sh, which was trying to upload to EOS also when not specified
- bugfix xml SHA and Version all zeroes in xml file
- (bugfix) keyword to start pipeline and resolve wip status is now <RESOLVE_WIP:>
- (bugfix) pick the correct version
- Release notes are automatically copied into the Official documentation
- CI automatically generates Release notes for stable and beta releases. Release notes are also automatically pushed to the documentation
- CI automatically generates Release notes for stable and beta releases. Release notes are also automatically pushed to the documentation
- update top.tcl, verilog and VHDL Templates

Beta Release v1.10.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F145%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 145
- Branch name: 103-checkyamlref-must-check-recursively-yml-file

Changelog

- check yml checks the SHA for the included yml files and all the local included yml files

Beta Release v1.11.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F138%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 138
- Branch name: 94-add-bitfiles-and-log-to-gitlab-releases-2

Changelog

- New tcl script to get binary files links for releases
- binary files can be downloaded directly from the release

Beta Release v1.12.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F146%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 146
- Branch name: 100-convert-getgitlabartifact-py-to-tcl

Changelog

- (bugfix) .gitlab-ci.ym file can now end with a comment

Beta Release v1.13.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F148%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 148
- Branch name: bugfix-wip

Changelog

- (bugfix) hog-child pipeline was starting also for WIP merge requests

Beta Release v1.14.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F149%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 149
- Branch name: 106-ipbus-xml-version-placeholder-is-replaced-with-hexadecimal

Changelog

- (bugfix): Version is added to xml files in M.m.p format

Beta Release v1.15.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F151%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 151
- Branch name: 107-getartifactsandrename-sh-problems-2

Changelog

- [bugfix] Fixed GetArtifactsAndRename.sh

Beta Release v1.16.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F129%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 129
- Branch name: planahead

Changelog

Beta Release v1.17.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F147%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 147
- Branch name: 101-use-shared-runners-as-often-as-possible

Changelog

- using shared runners when possible

Beta Release v1.18.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F152%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 152
- Branch name: sh-script-update

Changelog

- copy binary file to proper location on EOS
- Modified shell scripts to be fully compatible with supported HLS compilers

Beta Release v1.19.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F156%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 156
- Branch name: eosip-bugfix

Changelog

- (bugfix) Error is caused if a list file is included into another list file with a different extension.

Beta Release v1.2.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F132%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 132
- Branch name: fix-for-96

Changelog

Beta Release v1.20.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F158%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 158
- Branch name: 112-ssl-problem

Changelog

- work around for SSL CI failure

Beta Release v1.3.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F126%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 126
- Branch name: doxygen-config

Changelog

Beta Release v1.4.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F130%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 130
- Branch name: 95-smart-version

Changelog

- Option to specify library for list files included in other list files using the lib= property
- New function GetProjectVersion returns the last official tag when the project was modified or 0 if it was modified since last official tag
- pre-synthesis notices if the last commit for project is older than current commit
- when the repository is dirty the global SHA is still provided to the firmware, while the version is set to 0 to signal a dirty repository
- new utility script to calculate last SHA in which a project was modified:
- GetVer now only takes a list of files as argument as well GetHash renamed GetSHA
- Update and templates with new generics

Beta Release v1.4.1

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F135%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 135
- Branch name: hotfix-mkdir

Changelog

- (bugfix) Fixed bug in creation of a -p folder at post-bitstream stage

Beta Release v1.5.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F139%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 139
- Branch name: doxy-style

Changelog

- (bugfix) In case current commit is not contained in any tag, GetVer takes the last version from the last tag that is parent of the current commit, not the last tag in the repository as it used to be
- (bugfix) fix binfile was never created
- (bugfix) Fixed bug in creation of a -p folder at post-bitstream stage

Beta Release v1.6.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F134%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 134
- Branch name: 83-use-dynamic-yml-configuration-for-ci

Changelog

- New Hog option HOG_CHECK_PROJVER. If set to 1, the CI for a Hog project is run only if the project has not been modified with respect to the target branch
- hog.yml and hog-dynamic.yml includes the same yml config files (hog-main.yml and hog-child.yml), avoiding duplication of code
- (bugfix) fixed bug preventing Hog from creating a Gitlab release, in case one of the IP is modified during by the CI

Beta Release v1.7.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F141%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 141
- Branch name: bugfix-dynamic-ci

Changelog

- (bugfix) docker jobs run only on docker tagged machine
- (bugfix) fixed clean_unofficial script
- New Hog option HOG_CHECK_PROJVER. If set to 1, the CI for a Hog project is run only if the project has not been modified with respect to the target branch
- hog.yml and hog-dynamic.yml includes the same yml config files (hog-main.yml and hog-child.yml), avoiding duplication of code
- Option to specify library for list files included in other list files using the lib= property
- New function GetProjectVersion returns the last official tag when the project was modified or 0 if it was modified since last official tag
- (bugfix) fixed bug in LaunchIPSynth.sh, which was trying to upload to EOS also when not specified

Beta Release v1.8.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F143%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 143
- Branch name: bugfix-dynamic-ci

Changelog

- Simulation is now run in the same stage as the synthesis
- Restored external library support
- CheckYAML moved to merge stage

Beta Release v1.9.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F144%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 144
- Branch name: 104-official-tag-not-created-with-normal-pipeline-develop

Changelog

- xdc property set only one time
- (bugfix) restoring not merge-request pipelines for normal CI

Beta Release v2.0.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F161%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 161
- Branch name: before-script-fix

Changelog

Beta Release v2.1.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F162%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 162
- Branch name: bugfix-check-syntax

Changelog

- (bugfix) syntax was always checked, regardless the option

Beta Release v2.2.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F163%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 163
- Branch name: bugfix-bdfiles

Changelog

- fix typo in top.tcl template
- remove several info messages from GetRepoVersion function that would pollute the log
- unless HOG_NO_RESET_BD is set to 1, Hog-CI will reset .bd files in the pre-synthesis script
- VHDL address decode files can be missing from xml.lst file
- automatically reset files contained in VivadoProject/hog_reset_files, if it exists
- add GetModifiedFiles and RestoreModified files funztions. Restore modified bd files automatically in workflow launcher

Beta Release v2.3.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F164%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 164
- Branch name: feature-sim-runtime

Changelog

- Set do and udo to empty string as default
- option to set simulation runtime in .sim list file

Beta Release v2.4.0

<https://gitlab.cern.ch/api/v4/projects/74736/jobs/artifacts/refs/merge-requests%2F165%2Fhead/raw/changelog.md?job=changelog>

Repository info

- Merge request number: 165
- Branch name: bugfix-get-binary-links

Changelog

- fixing bug in get_binary_links.tcl when no binary files are available for the project

1.7.12 License

Copyright 2018-2020 The University of Birmingham

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

1.7.13 Authors

Francesco Gonnella (francesco.gonnella@cern.ch)

Davide Cieri (davide.cieri@cern.ch)

Nico Giangiacomi (nico.giangiacomi@cern.ch)

Nicolo Biesuz (nicolo.vladi.biesuz@cern.ch)

Alessandra Camplani (alessandra.camplani@cern.ch)
