Invited Paper: A Scalable Approach to IP Management with FuseSoC

Olof Kindgren
Qamcom Research & Technology, FOSSi Foundation
olof@fossi-foundation.org

***Abstract*—The lack of tooling to easily reuse cores and use them with new tools leads to unnecessary time being spent on manually adding tool support for each core, defining their relations and in worst case rewriting code because of the lack of a way to integrate existing code. FuseSoC rectifies this by bringing a modern package manager and a uniform build system to HDL developers. Having been around for seven years there are now hundreds of FuseSoC-compatible cores and 14 different simulation, synthesis and lint tools supported.**

This paper gives an overview of where FuseSoC can help with spending less time on the cores, and more time on the core business.

FuseSoC is an open source software project and available at https://github.com/olofk/fusesoc

***Index Terms*—IP, cores, manager, HDL, FPGA**

I. Introduction

FPGA/ASIC developers working on HDL code have a lot in common with software programmers. But at the same time, much of the infrastructure that software developers rely on has been missing in the HDL world. Two closely related such areas is package management and build systems [1]. The software developers have since long depended on package managers such as apt and yum for system packages, or language-specific package managers such as npm, cargo, pip or maven. The idea of a build system to provide an abstract interface for the target system is likely even older, with autotools, cmake, waf, scons and more recently meson as examples of such systems. Yet, for the HDL designers there has been no such commonly used tools. Many companies are using their own systems internally and the FPGA vendor each provide their own proprietary package formats. And instead of a common way to share code between projects, it is often the case that code is just copied around and modified locally, which makes it harder for improvements to reach all users and makes bugs live longer.

II. History

When working on ORPSoC v2 [2](the second version of the OpenRISC Reference Platform System on Chip) in 2010 this problem became very apparent as the number of supported configurations and included IP cores grew. Due to the monolithic nature of the project, every time a new addition or change was made, the whole code base was copied and eventually there was a large number of slightly different copies that had diverged from each other. Even more notable was that many of the IP cores that were initially copied into ORPSoCv2 were improved, but these changes were never made available to the original copy. When a new version ORPSoCv3 was planned, a number of goals were stated to improve the situation. Most notable of these were [3]:

- Focus on modularity
- Avoid changes to upstream cores
- Clear separation of source and generated files
- Easy to extend
- Easy to use
- Reuse existing technology

The work on ORPSoCv3 was started in 2011 as a Makefile-based project. After about a year of development, this turned out to be a dead end, and the project was restarted using Python instead. After another year or two of development, it became clear that none of the code in ORPSoCv3 was actually tied to OpenRISC, and that it was instead a general purpose package manager and build system for HDL. To make this more obvious the project changed name in 2014 to FuseSoC. The name itself comes from the metaphor of nuclear fusion where smaller cores are fused together to a large single core, as the IP cores were to be combined into a larger project. Development has continued continuously since then and as of December 2018 FuseSoC now supports 14 different EDA tools, has seen contributions from more than 30 developers and there are hundreds of IP cores packaged for use with FuseSoC. As the number of users grow, FuseSoC is aiming to become the main alternative for HDL package management.

III. Overview

FuseSoC itself is a command-line application and library written in Python that is used to perform various actions on cores. The cores expose different targets for running a test bench or implementing an FPGA image for a certain hardware device.

The command below will build an FPGA image for the *serv* core for a *TinyFPGA BX* board

```
$ fusesoc run --target=tinyfpga_bx serv
```

Some targets, often simulation, can be performed with different tools as can be seen in the following example of running the test bench of the *wb_bfm* core using *ModelSim*

```
$ fusesoc run --tool=modelsim wb_bfm
```

As can be seen from the above examples, both *target* and *tool* can have implicit default values.

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

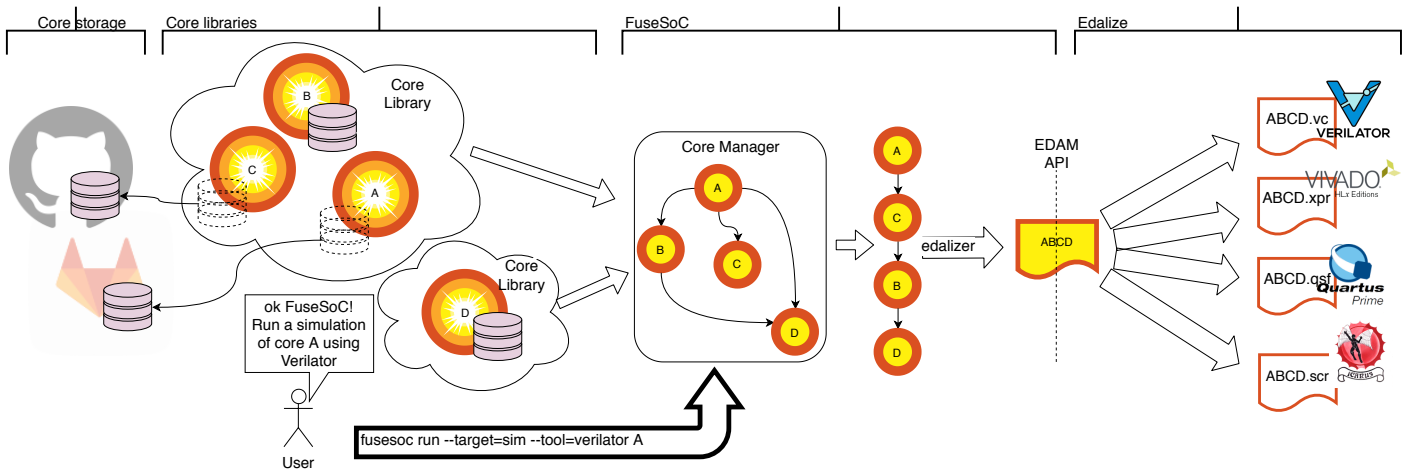

Fig. 1. FuseSoC is asked to run a simulation of a core A. The whole process consists of finding out which dependencies are needed and download them if required. The complete set of dependencies are then combined into a single design described by a tool-agnostic EDAM (EDA Metadata) object. The metadata is then converted into the project files needed to run the design through the requested tool and finally run the tool.

There are also actions for finding more information about the cores (*fusesoc core info*), working with generators (*fusesoc gen list*), adding libraries (*fusesoc library add*) and other aspects of the package management.

FuseSoC consists of several parts which provide different services, as illustrated in Figure 1. These can roughly be divided into:

- Core description files
- Dependency management
- Source / build separation
- EDA tool abstraction

To illustrate how these parts fit together, consider an example of a core *A* which has a target called *sim* for running a simulation and that the user wants to specifically use *verilator* [4] in this case. This would translate to the command *fusesoc run –target=sim –tool=verilator A*. When the above command is executed the following will happen internally in FuseSoC

1) FuseSoC's configuration files are read to find the location of the core libraries.
2) The core manager loads all core files from the core libraries
3) The core manager is queried to find all the dependencies of the requested core (*A*) given the specified target (*sim*)
4) With all dependencies resolved, there is now an ordered list of all required dependencies
5) Each of the core description files in the dependency list is parsed to see which files, parameters, VPI (Verilog Procedural Interface) libraries etc, are required for the requested operation. This information is combined to an *EDAM* [5] (EDA Metadata) data structure. All required files are also copied into a clean build tree.
6) Edalize [6], the EDA abstraction library, is handed the EDAM structure and proceeds to create the tool-specific files. As *verilator* was explicitly requested on the command-line (using the –tool argument), Edalize will proceed to create the project file and launch scripts required to build and run a simulation with *verilator*
7) After creating the project files, Edalize will continue to the *build* phase where the simulation model is built.
8) Finally, Edalize will launch the simulation model with any extra run-time parameters in the *run* phase.

IV. Core description files

Listing 1. Core description file example. This core description file describes a core with the VLNV identifier ::i2c:1.14. i.e. *name* is *i2c* *version* is *1.14* and the optional fields *vendor* and *library* are left out. The core has two filesets *rtl_files* and *tb*. Both contain only verilog files and the latter depends on minimum version 1.0 of the core *vlog_tb_utils*. The core has two targets. *default* is always used when a core is being used as a dependency of another core while the *sim* target can be used to run simulations on the core. Finally the *provider* section tells FuseSoC where to find the source code for the core and that it needs to apply a patch after it has been downloaded

```
CAPI=2:
name : ::i2c:1.14

filesets:
  rtl_files:
    files:
      - rtl/i2c_byte_ctrl.v
      - rtl/i2c_def.v:
          is_include_file : true
      - rtl/i2c_top.v
    file_type : verilogSource
  tb:
    depend:
      - ">=vlog_tb_utils-1.0"
    files:
      - tb/tst_bench_top.v:
          file_type : verilogSource

targets:
  default:
    filesets : [rtl_files]
```


This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

```
  sim:
    default_tool : icarus
    filesets : [rtl_files, tb]
    toplevel : tst_bench_top

provider:
  name : github
  user : olofk
  repo : i2c
  version : v1.14
  patches : [files/0001-add_vt_utils.patch]
```

The properties of a core are described by core description files. This description should be tool- and vendor-agnostic to ease the process of switching tools and targets. It should at the same time contain enough information so that the EDA tools can understand how to work with the core. On top of this, it should provide the user with information about the core and what can be done with it.

Listing 1 shows a basic example of a core description file. FuseSoC core description files implement a version of *CAPI* (Core API). To distinguish which *CAPI* that is implemented, files must begin with *CAPI=n*. This allows changing the file format completely between *CAPI* versions, something that was already done between *CAPI 1* and *CAPI 2*.

The rest of the file contains information about the core, such as which files it consists of, any parameters that can be set externally, EDA tool-specific settings or dependencies on other cores.

One defining feature of FuseSoC core description files is the optional separation between source and metadata. There are two types of cores and the type is determined by the presence of a section called *provider*, as can be seen in listing 1. A core with a provider section is called a *remote core* while a core without a provider section is called a *local core*. For a remote core, the provider informs FuseSoC where to find the actual source code. The first time the core is needed, FuseSoC will download the core source code and store it in a local cache and use that location for subsequent uses. For the local cores FuseSoC will instead look for the source code in the directory where the core file itself resides. The two types exist to support different workflows. A local core is commonly used when interacting directly with the core, such as during development. This allows the core description file to be developed and rapidly changed in tandem with the source. A remote core on the other hand is useful for building a library of third-party cores without having to make actual copies of all the cores. Being able to store the core description file separate from the source code is also especially important for cases where it's not allowed to host a copy of the source code. In these situations it might also be required to make changes to the third-party source code, for which FuseSoC allows applying patches to remote cores stored in the cache. Figure 2 show the transitions of the source code for remote cores.

The idea of using a file to encapsulate properties of a core

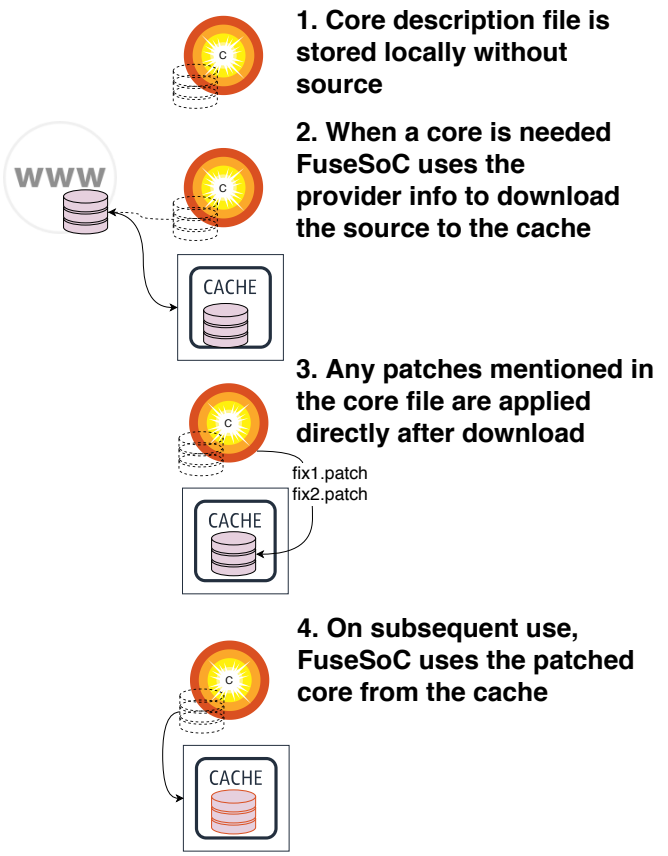

Fig. 2. FuseSoC cache flow for remote cores

is not unique to FuseSoC. Most notable is IP-XACT [7] but there are also vendor-specific formats such as Xilinx XCI (which effectively are Xilinx-branded IP-XACT 2009 files) or Intel's Qsys [8]. There are also a large number of package formats for software, such as deb [9], RPM [10] or ebuild [11] packages. Reusing an existing package format has the benefits of being able to reuse existing knowledge and tooling. There are however issues with all the aforementioned formats that makes them less ideal for the purpose of FuseSoC.

Starting with the vendor-specific formats, they are tied to a single vendor which defeats the idea of a description format that all tools can consume.

The software package formats tend to focus on distributing and installing binary payloads that are compiled by the package provider and distributed in binary form to be installed by the users. This isn't suitable for HDL where the analogous binary format would be a netlist synthesized for a particular device family or a part of a simulation model for a particular tool. While a software package needs to be available for one or mostly a handful of architectures (e.g. x86 and arm), the HDL package would need to be built for multiple version of multiple tools and multiple hardware targets which makes a portable binary distribution infeasible.

There is a class of source-based package formats that work by distributing source code and builds the binaries on the users

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

machines. Gentoo's ebuild or Rust's cargo [12] are examples of this. The source-based package formats remove the distribution problems, but there are other issues that still make them unsuitable for HDL package management due to differences in the software and HDL ecosystems. Existing source-based software package formats usually contain instructions for the steps involved to transform the source code to an executable or library on the user's system. These steps generally involve *configuration*, *compilation*, *test* and *installation*. It is also assumed that a particular tool chain, or occasionally two (e.g. GCC or LLVM) is used. HDL on the other hand is consumed by a variety of tools with different purposes and the steps can instead be *elaboration*, *synthesis*, *compilation*, *solving* or *STA*. With the variety of tools also come many tool-specific file types that are only applicable for that tool (e.g. vendor IP core descriptions, memory initialization files). All this adds up to make existing software package formats a bad fit for HDL.

The existing format that comes closest to fulfil the requested traits is IP-XACT. This was created as a *Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows* [7]. The standard is too large to cover in detail, but it contains structures for describing the files, external connections, register maps, parameters and many other things that makes up an IP.

There are several areas of IP-XACT that overlaps with FuseSoC core description files and the latter are influenced by the former where applicable. The two most notable examples would be that a core in FuseSoC is uniquely identified by a VLNV (Vendor Library Name Version) identifier as in IP-XACT. The notion of file sets and their attributes (e.g. *is_include_file*, *logical_name*, *file_type*) is directly copied from IP-XACT and the enumerated list of supported file formats is a super-set of the valid IP-XACT file types.

This would imply that IP-XACT could be used as is instead of creating something new. There are several reasons why it was deemed necessary to create a new core description format. As of the current version of the standard at the time of writing, IP-XACT is missing features which FuseSoC relies on to perform its tasks.

While IP-XACT has a notion of dependencies between cores it lacks support for version ranges, a feature which exists in effectively all package managers, and without which it is difficult to build dependency chains.

IP-XACT also enumerates valid file types and EDA tools. Both are missing values that are required to provide sufficient guidance for the tools. As an example, IP-XACT defines the file types *vhdlSource*, *vhdlSource-87* and *vhdlSource-93* but is missing an entry for e.g *vhdlSource-2008* which is vital information for some tools that treat these types of files differently.

To mitigate the issues of missing features, IP-XACT has from the beginning implemented a system for *vendorExtensions*, a facility to add non-standard features to an IP-XACT description file. Vendor extensions would likely be able to solve the above issues, but one issue remain that is not fixable this way.

IP-XACT files are not suitable for direct manipulation by humans, meaning that additional tooling is required. FuseSoC core description files are intended to be easy enough to manipulate by hand while providing enough information about the core to guide the EDA tools. This doesn't mean that FuseSoC core description files replaces IP-XACT. There are several areas where IP-XACT provide information that FuseSoC has no interest in, but are still vital for other uses of the IP core (e.g. register maps and interfaces). Fig. 3 show a Venn diagram of FuseSoC core description files and IP-XACT files.

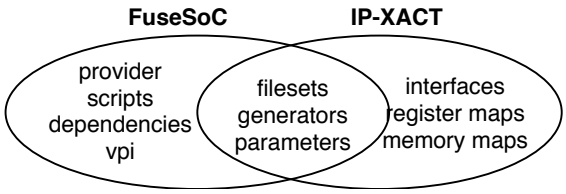

Fig. 3. Feature comparison of FuseSoC core description files and IP-XACT files

For the areas that overlap between FuseSoC and IP-XACT, FuseSoC is in certain situations capable of extracting this information from the IP-XACT file instead of having to duplicate it in the core description file.

V. Dependency management

To facilitate reuse of cores or subsystems in a SoC, there must be a way to specify what to reuse. This is the task of the dependency manager. A core can be depend on zero or more other cores. The set of dependencies can be different if the core is used as a part of a larger system or if it is itself the top-level core; whether it is being used in a simulation or synthesis context. Dependencies doesn't have to be an exact version, but can be specified as a range or any version. The use of version ranges is important to minimize conflicting dependencies (e.g. the *Diamond dependency problem* [13]) and is used in essentially all software package managers, but is notably missing in other HDL-specific equivalents such as hdlmake [14], IP-XACT or IPApproX [15]

The complete tree of dependencies is fed into a SAT solver [16] that processes the requirements and provide an ordered list of cores which satisfies all constraints.

VI. Source / build separation

A complete design can consist of a large number of cores. This makes it harder to know exactly what files and settings were used for a particular build and how to reproduce it.

FuseSoC solves this problem by separating the build process into several steps. First all files to be used for the build are transferred to a new build tree so that the files used for a build are protected even if the underlying source changes. In the next step when the EDA project files are created only files in the build tree are referenced. This is called the *configure* stage. FuseSoC can be instructed to stop after this step so that the build tree can be archived for traceability or transferred to another machine that handles the build. As no tools are called

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

during the *configure* stage, this can all be done on a machine without any EDA tools installed, making it easier for a team of developers to work on their local machines and submit job requests to remote build machines.

VII. EDA TOOL ABSTRACTION

FuseSoC supports more than a dozen different EDA tools. Yet each of these tools use their own unique format for listing input files (e.g. HDL source, constraint files, memory initialization data) and applying settings. A single design is often used by several tools. There could be one tool for synthesis/P&R, one for formal verification and several types of simulators. These tools will typically consume slightly different subsets of the source. As for simulating the same test bench with different simulators, they often use the exact same subset of the design source. At the same time, each new tool to be used requires creating a whole new project file listing mostly the same HDL source, top-level module, compile-time configuration (e.g. verilog ̀define, verilog parameters, VHDL generics) and run-time configuration (e.g. plusargs).

FuseSoC solves this by defining a metadata format called *EDAM* that describes most parts of the design in a tool-agnostic way while leaving room for applying tool-specific directives. The *EDAM* description is then passed through a tool-specific back-end to be transformed into the project files required for each EDA tool.

Once the project files are created, FuseSoC can be asked to run the tool. Running a tool has different meanings depending on the type of tool. For simulation this generally consists of compiling and running a simulation model. When creating an FPGA image the steps can be synthesis, placing, routing, STA, image generation. A linting tool typically only has a linting step. While other tools such as HAMMER [17] and hdlmake [14] provide fine-grained abstractions to run each of these steps, FuseSoC has chosen the path of only providing two steps - *build* and optionally *run*. The *build* step will do transformation of source code to a binary representation, while the *run* step will execute what was created in the build phase. For a simulator/formal verification tool this means that the build step will create the simulation model while the simulation/solver runs in the *run* step. An FPGA tool chain will create the FPGA image in the *build* step and leaves the *run* step undefined. The rationale for having a coarse-grained approach is that most EDA tools are complex and support many different workflows and as more EDA tools are supported, more different steps would need to be defined as they might not fit into the existing ones. It is instead assumed that the user will use the FuseSoC-created project files and launch the EDA tools themselves for any workflow not directly supported by FuseSoC.

VIII. CONCLUSION

This paper show several areas where FuseSoC play an important role in filling a gap in the current EDA ecosystem.

Being able to reuse existing components in a new design is important to reduce to cost and effort of the new design. FuseSoC can help here by allowing users to build up core libraries where open source and proprietary, in-house developed and third party cores can be combined.

Being able to use cores in a new environment is important to allow them to be used by users with access to different tools than the original authors. FuseSoC can help here by providing abstractions for EDA tools to make it easier to change.

Having been around for seven years and used in many designs, FuseSoC has stood the test of time. FuseSoC continuously expands its support for new EDA tools and other features as well as finding an ever increasing supply of FuseSoC-compatible cores. As FuseSoC is open source software (https://github.com/olofk/fusesoc) anyone is free to use and contribute to the project. The FuseSoC source code is available under the GPLv3 license. This means that any changes to the FuseSoC code base must remain under the same license. The Edalize library is available under the BSD license which allows it to be embedded into other open source or proprietary products without necessarily remain open source. Core description files are considered as configuration files and not covered by any license. The core description files found in the FuseSoC base libraries [18] [19] are freely available to use and modify. Finally, any files created by FuseSoC or Edalize (e.g. EDA tool project files, Makefiles, run scripts) are also not similarily not enforced any license, and the user who invoked FuseSoC to create the aformentioned files is free to choose their own license for them.

IX. FUTURE WORK

- *Use flags*: FuseSoC already support setting compile- and run-time parameters to parametrize the HDL source, but there is also a need for a mechanism to conditionally enable and disable features in the core description files. This can be compared to Gentoo's USE flags [20] or Rust's conditional compilation [21]. This is already partially implemented in FuseSoC and can be used to conditionally add or remove filesets depending on which tool that is used, but lacks a way for users to define and set custom flags.
- *Better IP-XACT integration*: Given the number of features that overlap between FuseSoC core description files and IP-XACT it should be possible to use more features directly from the IP-XACT descriptions. Also to investigate is the possibility of adding features from FuseSoC core description files to future IP-XACT revisions.
- *Lock files*: When the dependencies are resolved for a core, it will by default use the latest available version that satisfies all solver constraints. If there are updates to a sub-dependency, this new version will likely be used during the next build. In some cases it is instead beneficial to build a design with the exact same dependencies even if there are newer versions available. Lock files is a way to handle this. It is a file that states the exact version to use for each dependency, which will override the regular dependency calculation.

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

- *Increased tool support*: FuseSoC currently support 15 different tools, but there are several simulators not supported, e.g. *NCSim*, *NVC*, *ActiveHDL*, and FPGA tool chains missing, e.g. *Diamond*, *Libero*. There are also other types of tools missing such as *SymbiYosys* [22] for formal verification.

ACKNOWLEDGMENT

I would like to thank all users of and collaborators on FuseSoC. The code base currently holds contributions from more than 30 different people. On top of this are a large number of users who have provided guidance in form of bug reports or by explaining their workflows to provide input on what to improve. Thanks also to everyone who have worked on adding FuseSoC-compatible cores to expand the ecosystem. You know who you are, which is more than I know

REFERENCES

[1] "Build automation" https://en.wikipedia.org/wiki/Build_automation
[2] "ORPSoCv2 (OpenRISC Reference Platform System on Chip version 2" https://github.com/openrisc/old-openrisc-svn/tree/old-svn/trunk/orpsocv2
[3] "ORPSoCv3 - Solving the core issue" https://www.youtube.com/watch?v=vYJjIoV0G3U
[4] "Verilator" http://veripool.org/projects/verilator
[5] "EDAM" https://edalize.readthedocs.io/en/latest/edam/api.html
[6] "Edalize" https://github.com/olofk/edalize
[7] "IP-XACT" https://www.accellera.org/downloads/standards/ip-xact
[8] "Qsys System Integration Tool Support" https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-software/qsys.html
[9] "Debian Packaging" https://wiki.debian.org/Packaging
[10] "RPM Package Manager" http://rpm.org
[11] "ebuild" https://wiki.gentoo.org/wiki/Ebuild
[12] "The Cargo Book" https://doc.rust-lang.org/cargo/index.html
[13] "Dependency hell" https://en.wikipedia.org/wiki/Dependency_hell#Problems
[14] "hdlmake documentation : Handling remote modules" https://hdlmake.readthedocs.io/en/master/#handling-remote-modules
[15] "IPApproX - Set of IP management tools used within the context of the PULP project" https://github.com/pulp-platform/IPApproX
[16] "SAT solver" https://github.com/enthought/sat-solver
[17] "HAMMER: Highly Agile Masks Made Effortlessly from RTL" https://github.com/ucb-bar/hammer
[18] "Core description files for FuseSoC" https://github.com/openrisc/orpsoc-cores
[19] "FuseSoC standard core library" https://github.com/fusesoc/fusesoc-cores
[20] "What are USE flags" https://wiki.gentoo.org/wiki/Handbook:AMD64/Working/USE
[21] "The Rust Reference : Conditional compilation" https://doc.rust-lang.org/reference/conditional-compilation.html
[22] "SymbiYosys (sby) – Front-end for Yosys-based formal verification flows" https://github.com/YosysHQ/SymbiYosys

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io