CHES Hardware Artifact Submission Tips and Suggestions

Ben Marshall

Abstract: This document serves as guidance on how researchers and engineers can package their hardware projects as part of the CHES Artifact Review Process. It is designed to capture best practice when it comes to improving the re-usability and reproducibility of hardware projects in the cryptographic research community.

Introduction

The main aim of this document is to provide guidance to engineers and researchers submitting their hardware projects to CHES Artifact Review Process on how best to document and package their project to enable others to reuse it, and replicate their results. A side effect of this is to help the cryptographic research community produce the highest quality research artifacts possible, and provide guidance on good hardware engineering practices, even if you aren't taking part in the CHES Artifact Review Process.

This document focuses on projects which submit hardware descriptions and designs as part of their artifact. By this, we usually mean hardware description languages (like Verilog & VHDL), which are intended to be implemented as part of an ASIC or FPGA design.

Specifically, it tries to:

- List what is required in a good submission, including things beyond just the RTL source code.
- List easy things people can do to build *high quality* hardware artifacts.

- Inform people on how to package their projects to make it easy for others to reproduce their results.
- Inform people on *why* these requirements matter.

The best time to be aware of all of these things is *before* you start writing code, so you can plan your project more easily and avoid retrofitting best practice onto your project. However, most of these things can be added on top of an already completed project without too much effort.

Ultimately, our community does a lot of *engineering*, as well as *research*. High quality engineering is a foundation for good research, just as good research leads to better engineering.

Before you read any further Make sure you are familiar with the CHES Artifact Review Process as described on the CHES website. This guide is an extension of that, aimed at making everyone's life a bit easier.

CHES Specific Guidelines

This section lists a set of guidelines which, if met, would lead to a high quality CHES artifact which is valuable to the community, and impactful in terms of its reusability. These things are generally applicable to other hardware projects, but emphasise things which have come out in past CHES artifact review cycles. It is split into guidance for general hardware projects, and some specific recommendations for FPGA or ASIC focused projects.

General Hardware Projects

The RTL Source Code

It's hard to have a hardware artifact without this. This is typically your (System)Verilog or VHDL code (or even a more recent HDL like Chisel, or a Python based DSL like nMigen).

- The RTL code should follow good practice in how it is written. You may wish to follow a style guide which can help with readability and maintaining quality. The LowRISC Verilog style guide is a good and comprehensive example, though you can improve your code immensely by following only a subset of this.
- The RTL must be well commented (especially port definitions). For example, is your "`operation_finished`" signal a *level* trigger or a *pulse* trigger? Which clock is it synchronous too? Etc. This is essential information for anyone looking to reuse your code.
- It should be well structured into modules, with common modules (e.g. your Keccak Implementation) self-contained and easy to reuse.

    - This structure should be documented. It's much easier for reviewers to navigate source code if we know which modules instance which other modules, for example. A block diagram or simple hierarchy tree is a nice way to show this.
- It is necessary to document which tools and platforms your design is *expected* to work on, including version numbers for those tools. E.g. If you directly instantiate DSP blocks, your design might only be compatible with that specific FPGA vendor's simulator. Providing *functionally* alternative ways to instantiate vendor specific blocks will drastically increase your projects reusability. If nothing else, you should make sure vendor or platform specific code is clearly labelled or separated from generic or platform independent code.

NOTE: EDA tooling is famous for having differences in accepted syntax between Vendors. E.g. what is acceptable to Xilinx Vivado may not be acceptable to Synopsys VCS, or Icarus Verilog. The same is true for Vendor specific pragmas or attributes, which might be ignored by other vendor's tools and affect the results. In the author's experience, Verilator is an extremely good & easy tool for linting your (System)Verilog code to make sure it is acceptable to the widest range of other tools. It also picks up width mismatches between signals which Vivado will happily ignore. Any vendor specific pragmas should be listed clearly for reviewers and future users of the code.

- Likewise, if your design contains components which cannot be simulated well (e.g. a ring oscillator), providing a simulation-only variant of that functionality, which can be substituted for the real thing during implementation is very helpful.

Testbench

You *must* include a testbench with your RTL code.

- The testbench must demonstrate that your design performs as described in your paper, and convince the reviewers that your design is *functionally* correct.
    - For example, a handful of test vectors is not enough to convince a reviewer your cryptographic accelerator design is correct. Good practice for this is to use standard Known-Answer-Tests (KATs) (e.g. NIST test vectors for their algorithms) or generating a large number of test vectors for not-yet-standard designs.
    - You can build confidence you have enough test vectors by using code-coverage metrics to check that you have exercised the critical parts of your code. All commercial simulators support this, and the open source Verilator simulator also supports code coverage collection and reporting[1].

1 In commercial hardware development, one would also typically perform functional coverage driven verification.

    - You might also include some formal verification[2] to prove that certain critical parts of your design are correct or behave as expected. The Symbiyosys tool is an excellent free and open source formal verification suite. For example, you might assert that after your modular reduction, the result is always in the correct range.
- The testbench must include some form of automated checking (whether this is inside the HDL, or as a post-processing step to compare log files). Manual checking is boring and prone to error. This makes it much easier for others trying to improve your design to know they haven't broken it.
- The testbench must come with instructions on how to run it. Running the testbench must be automated using a script or (ideally) a Makefile. Likewise, instructions must be provided in your README or user-manual. These instructions must include the versions of the tools needed to run the testbench.
- It is helpful to describe what reviewers should *expect* to see when looking at waveforms and running the testbench generally. This includes:
    - *Where* waveforms, logfiles, bitstreams and implementation reports (if any) are written to.
    - What features to look for in waveforms which indicate correct / expected behavior.
    - How to interpret output reports so they can be matched with results in the paper.

NOTE: The importance of functional verification of hardware artifacts in the academic community is often under-emphasised. It is hard for artifact reviewers to be confident a design is correct without a comprehensive and well-justified testbench.

Documentation

- Having a dedicated *user guide* which contains all of the *engineering* information (as opposed to what is academically interesting, which is what your paper is for) is always a mark of a good artifact.
    - When in doubt, the LowRISC documentation is always a good example of what to aim for. E.g. their AES co-processor documentation. Again, you don't need to go into *this* much detail to massively improve your own project's reusability and value to the community.
- Describing the main interfaces to your design (e.g. when data signals can be sampled, are there timing constraints on the control signals) makes it *much* easier for others to reuse your code. Otherwise, these things must be reverse engineered.
- A block diagram is worth 10 pages in your academic paper.
- Timing diagrams created using Wavedrom are also worth 1000's of words.
- Describing how your project is organised, including where the major components are located in the repository is very helpful to reviewers.

2 Note this refers to formal functional verification of hardware behavior. Not a formal proof of security for something like a masking scheme or cryptographic system.

Flows, Tooling and Ancillary Components::

Hardware projects consist of a set of "flows". These are as integral to the artifact as the source code, and should not be neglected.

- The simulation flow for checking the designs functional correctness and performance characteristics.
- The implementation flow for checking resource utilisation or operating frequency results. This flow often consists of several steps: synthesis, place and route, design rule checking, power estimation etc.
- Post-implementation evaluation. This might include scripts to run the design on an FPGA and acquire actual performance numbers or run correctness tests, or to extract power profile information for side-channel assessments.

These flows often have inputs other than the RTL source code:

- Constraints to describe placement and timing rules.
- Scripts to configure parameters inside the tool flow (e.g. optimisation goals).

It is necessary to document all of your projects flows, and make them as easy to reproduce as possible. The easiest way to do this is to automate them with scripts and/or Makefiles, and describe how to drive these scripts in your user-manual.

FPGA specific Projects

FPGA projects can often be made perfectly reproducible for reviewers by including a project generation script. For Xilinx Vivado, one can recreate this using "`File -> Project -> Write TCL`", and save the resulting TCL script to the project repository. Including instructions for running Vivado to create the project from this TCL script in your README makes it trivial for reviews to replicate your exact setup.

FPGA projects must also include constraint files, describing any placement or timing constraints on the design.

FPGA projects should also include a harness or supporting environment to actually run the design on the FPGA. For example, a cryptographic IP cannot be implemented directly onto the FPGA, and needs supporting logic to e.g. communicate with a host PC, and supply inputs / retrieve outputs.

In the past, some artifacts have been received with no such harness to actually run the design on an FPGA. This is problematic on several levels:

- In some cases it leads to the ports of the top-level module in the design being auto-assigned to actual FPGA pins, which would never normally occur in a real design. This

distorts timing results due to unrealistic cell placement and path lengths, and can even distort how tools place cells within the design.
- The practice of integrating a design into a wider system can show up bugs or inefficiencies which would otherwise go unseen. Examples include un-expected feed-through paths and combinatorial loops, or design choices which make interfacing with a wider system more or less efficient than expected.
- Having a harness which allows actual running on an FPGA makes it possible to distribute bitstream files to other researchers and reviewers. Without this, certain kinds of results (like side-channel power measurements) cannot be reproduced.

While the inclusion of such a harness is slightly more work for researchers, it is still essential as a way of building confidence that the design is feasible to integrate into a wider system. This is particularly important for cryptographic hardware, which is almost always facilitating the correct / secure / authentic behavior of a larger system. Ultimately, claiming to build an FPGA project which has not been properly run on an FPGA falls short of the expectations for the CHES Artifact Review Process.

FPGA project artifacts should also include a generated bitstream. Not only for the reasons above associated with the harness, but to make it easy for researchers to experiment on *exactly* the same design used in the original paper (in case exact tool versions to re-generate the project are no longer available). This is particularly important for power and EM side-channel related works.

ASIC specific Projects

ASIC project artifacts (i.e. those targeting a particular technology node and standard cell library) should also include the following to be as easy to replicate as possible.

- Instructions for obtaining the technology files and standard cell libraries used for the project. If these are not freely available, then enough information should be provided such that anyone who is in a position to purchase access knows what to ask for.
- Exact tool information, including version numbers. If open source tooling is used, or can be used alongside commercial tooling to produce representative results, (e.g. using Yosys and OpenROAD) this makes the reviewers task of building confidence in reported results much easier.

NOTE: While commercial tooling is sometimes available at little or no cost to *academic* institutions (through programs like EUROPRACTICE), not all institutions or researchers will have access to these programs. Hence it can be useful to use open source alternatives *and* commercial versions of these tools. In the long term, this makes it considerably easier to directly compare results across the literature. Open source EDA tooling has improved dramatically in recent years, and the cryptographic community has a lot to gain by embracing the benefits it brings.

- Necessary scripts to drive the tools and replicate any implementation flows should also be provided. These can be made easier to use by wrapping them with Makefiles.

General Tips & Wider Reading

- FuseSoC is a “Package manager for hardware”, and tries to automate the packaging of hardware projects and certain flows.
- Edalize is “a Python Library for interacting with EDA tools. It can create project files for supported tools and run them in batch or GUI mode”.
- There is a wide variety of open source hardware tooling which can make developing and sharing your hardware project much easier for the community.
    - Verilator - A very fast (System)Verilog simulator with excellent linting support.
    - Icarus Verilog - Another great Verilog simulator with support for 4-state simulation.
    - GHDL - An open source VHDL simulator.
    - GTKWave - An open source Waveform viewer.
    - Yosys - Open source synthesis tool, with support for various FPGAs and standard cell libraries.
    - Symbiyosys - Open source formal verification tools and flow. Really easy to use.
    - OpenROAD - “an automated RTL to GDSII flow [...] The flow performs full ASIC implementation steps from RTL all the way down to GDSII.”
- Best Practices For Sharing FPGA Designs from the Open Source Hardware Association provides a great list of things to consider when sharing FPGA projects generally. It provides a particularly good list of other things you need to provide other than HDL source files, and how to license your source code to make it easy for others to use.