ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-TUAPP01

HARDWARE-IN-THE-LOOP TESTING OF ACCELERATOR FIRMWARE*

C. Serrano†, M. Betz, L. Doolittle, S. Paiagua, V. K. Vytla, LBNL, Berkeley, USA

Abstract

Continuous Integration (CI) is widely used in industry, especially in the software world. Here we propose a combination of CI processes to run firmware and software tests both in simulation and on real hardware that can be well adapted to FPGA-based accelerator electronics designs. We have built a test rack with a variety of hardware platforms.

Relying on source code version control tools, when a developer submits a change to the code base, a multi-stage test pipeline is triggered. Unit tests are run automatically, bitstreams are generated for the various supported FPGA platforms and loaded onto the FPGAs in the rack, and tests are run on hardware. Reports are generated upon test completion and notifications are sent to the developers in case of failure.

OVERVIEW

FPGA-based accelerator instrumentation is based on the coherent design of firmware, controls software, hardware and communication links in between. All those layers are usually not static entities but continuously evolve to provide more features or improve performance. In addition to this coherence problem, FPGAs are becoming larger and more complex and, as software, firmware designs follow a fairly complex layered approach in themselves.

In this paper we describe some of the practices we have found useful at Berkeley based on common version control tools and CI setup. Adopting these processes was a rewarding experience and proved easier to deploy than it seemed at first. It has facilitated the task of developing complex, layered designs where these layers are tightly coupled and need to be tested in conjunction [1]. When one of us is working on a particular layer in the design and commits a change in version control, the CI setup automatically runs unit tests, builds FPGA bitstreams and runs fully integrated tests on hardware, verifying that the entire design works as expected as a whole.

Although, we have been using version control software and been writing several tests for our firmware and software designs, testing on the bench, or updating firmware on deployed electronics usually involved verifying many changes all at once. With the CI process in place, not only can we catch failures more often but the process is automated and if something goes wrong after committing a change the developer who made the breaking change is setup to be notified by email.

* This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-76SF00515.
† CSerrano@lbl.gov

MOTIVATION

While our group in particular handles specific technical challenges and integration issues [2], the general development processes can be extended to those in other fields of accelerator instrumentation. Those include:

- Growing complexity of multi-layered designs.
- Issues related to interconnection of design layers and overall coherence and integration.
- Collaborative development among organizations where different groups, sometimes in separate institutes, are responsible for layers within a common design.
- Lack of quality assurance and automated testing processes.

Continuous integration is a development practice used to prevent integration problems while modifying a small part of a larger design. Industry practices have in the past forced developers to run unit tests locally before committing changes to shared repositories. Today, this process has been widely adopted and some of this tedious testing and verification steps have been streamlined and automated.

Automated verification and continuous integration practices become particularly important when dealing with complex systems with many integrated parts, especially when multiple groups work together on a design or parts of those designs are shared among different projects.

The issues related to design complexity are not at all unique to accelerator instrumentation but are in fact common in electronics and software industries [3]. However, instrumentation and controls groups in accelerator facilities usually lack solid development process and quality control, let alone automated testing processes.

Deployment is also a very important step in accelerator electronics, usually carried our by software controls groups who have more established release and deployment processes. A usual example of integration difficulties encountered in accelerators is guaranteeing the coherence between FPGA firmware designs and their associated software low-level drivers and high-level applications. If an FPGA designer adds or removes a feature-or even renames a register that is expected by controls software-there needs to be a process for those parts of the design to stay in sync. CI processes help facilitate those steps by testing the entire system as a unit and by providing "artifacts" readily available to all parties. These artifacts can be FPGA bitstreams or software executables which are automatically generated by the CI tools every time a developer submits code changes. These artifacts can be set to be deployed automatically as well.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-TUAPP01

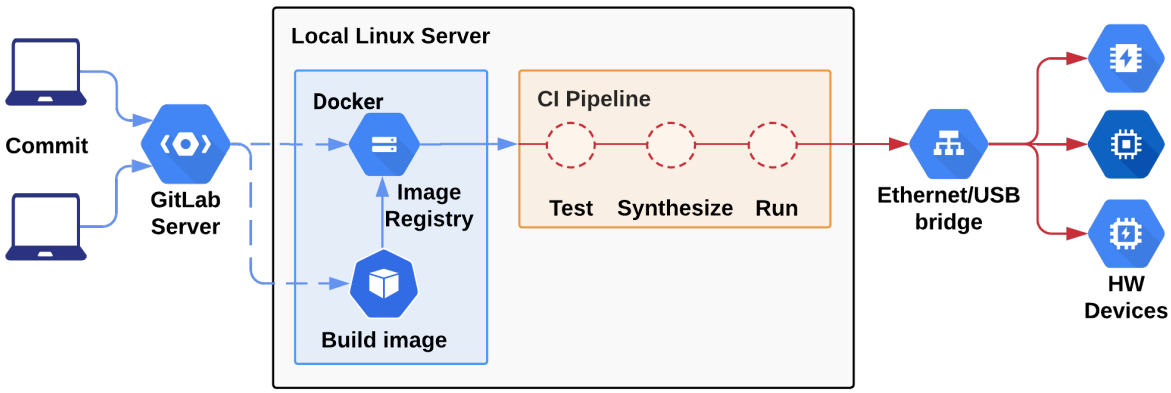

Figure 1: Hardware and software CI flow.

In the following sections we dig deeper into how these processes have been established at LBNL and provide some details on our specific implementation. These concepts can be applied to other needs and implemented using different tools or frameworks.

CONTINUOUS INTEGRATION

Continuous integration (CI) builds upon the strengths of Version Control Systems (VCS) to create a more efficient and robust development flow. Version control facilitates the collaboration between multiple developers working on a common code-base by providing a means to merge, revert and compare changes. However, errors introduced inadvertently can go unnoticed for multiple revisions, which can have a compounding effect and greatly increase debugging effort. Moreover, as systems grow in complexity, it becomes more challenging for a developer to anticipate what side-effects his/her changes might have elsewhere.

The use of CI mitigates these issues by introducing a quality assurance (QA) step that is automatically triggered on every commit, in the form of a suite of checks and tests that aim to quickly measure the impact of the change being made. This early feedback makes it easier to identify functional regressions and address them before they've propagated. Additionally, the inclusion of code quality and style checks in the automated test suite can be an efficient way of keeping the quality of a codebase high.

It is important to note, however, that the benefits reaped by employing CI will only be as good as the automated test suite it is using. Therefore, the first step towards the effective deployment of CI in a development flow is to carefully plan and create a collection of tests that adequately capture both the individual and integration requirements of the various building blocks of the project.

Creating a Test Suite

While it is important to be mindful of the overall execution time of a CI test suite, there is no real limit to what it can encompass. Indeed, any check or test that can be reduced to a well-defined sequence of steps and be made to return an appropriate exit status is a candidate. Despite how flexibly tests can be defined, it is advantageous to have a build system in place, such that common build, test and run steps can be triggered with a single-command. Having access to this capability greatly simplifies the creation and management of a CI pipeline.

Individual tests or checks within a CI suite are typically referred to as jobs. What follows is an example list of the types of jobs that we have successfully integrated into our automated testing pipeline:

- Run linters, such as Python PEP8 compliance checking for code readability
- Verilog linting checks with Verilator [4]
- Compilation of Verilog source code with Icarus Verilog [5] and Verilator
- Running of self-checking testbenches
- Synthesis of HDL with Xilinx Vivado
- Bitstream generation and timing closure checks
- Live hardware tests on a wide range of FPGAs

In addition to pass or fail results, individual jobs can be configured to store artifacts, consisting of arbitrary collections of files generated during its execution. This can be a very useful way to archive the results of long-running processes, such as FPGA bitstream generation, as well as to keep detailed logs of hardware tests for later triage/analysis.

Finally, it is important to note that the definitions of the CI suite and pipeline are files that are themselves kept in revision control. This means that any improvements and changes made to the testing process can be reviewed, merged and reverted just as any other piece of source code.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-TUAPP01

Architecture

The architecture of a CI system (as shown in Fig. 1) can be as simple or as complex as the project it is being used with. As an area of active interest, especially in the software development industry, several technologies and tools exist that address and enhance the various parts that make up such an automation system. These vary greatly in complexity, ease-of-use and learning curve but, in its essence, a modern CI environment can be distilled down to three basic elements:

- A CI Server that monitors a version-controlled code repository for changes
- A dependency management system
- A "runner" that executes the tests and reports their status to the CI server

Whenever a developer commits a code change to the project repository, the CI server is initialized. After obtaining a complete snapshot of the updated codebase, it must prepare the environment in which the tests that comprise the CI pipeline will be run.

This step is especially important because it allows tests to be run in a deterministic environment that doesn't change over time and greatly contributes to the reproducibility of results. This is in contrast with tests that are manually initiated by developers in their own machines, where different tool versions or local untracked changes may exist, thus affecting the results in unexpected ways. The ability to explicitly and uniquely specify the dependencies and environment used by the CI test suite is therefore key to maximizing the benefits such development flow can bring. Here, too, multiple software offerings exist that help tackle this challenge, typically involving a form of containerization, where a full runtime environment is generated on demand.

Once the environment is initialized, a "runner" process takes over and starts executing the test suite. Related tests can be aggregated in stages, within which, the jobs running each test may progress in parallel. Later stages depend on earlier stages completing successfully, creating a dependency system that can be leveraged to organize the pipeline in an increasing degree of complexity and execution time, so that bad commits fail as early as process in the process, saving test cycles and reducing feedback times.

The "runner" may be run on any network-accessible server machine. We have found containerization (e.g., Docker, chroot) helpful for documenting and isolating the tools and their valid versions. Also, such containerized processes can easily be run on any machine, independent of its native software environment. A notable exception occurs when the CI pipeline includes tests that require direct interaction with specific hardware devices that are not, or cannot be accessed from the normal network. In such cases, the "runner" necessarily runs on the properly hardware-connected machine. Security concerns often mandate such a setup even when the devices under test are Ethernet-connected, so they can be kept isolated on a private network,

DEVELOPMENT & VERIFICATION FLOW

Simulation is always an essential part of FPGA development. Along with HDL code, testbenches get written, starting with simple unit tests. Our CI framework builds on that reality, starting with the mandate that developers not only write testbenches such that the waveforms look right, but also produce a PASS/FAIL result. These testbenches can become stricter with time and experience.

Our Verilog-centric experience involves mostly Icarus Verilog run from GNU Make. This "make checks" step is then easy to encapsulate in a CI environment running shell within Docker, so that these tests are accumulated and run systematically.

Firmware on the FPGAs pretty much never runs in isolation. When it interacts with software, it's important to simulate the two together, a process often described as hardware-software co-simulation. Techniques vary, depending on the communication framework between the FPGA and software. When the software runs on a soft core inside the FPGA, ordinary HDL-style simulations can work without conceptual modification. Compilers are added to the required infrastructure, and the binary output gets loaded by the simulation. In contrast, programs running remote to the FPGA normally need a simulation of the data transport channel. Then the software can run either natively or in a dedicated emulator.

Our experiences cover RISC-V [6] soft cores, and C and Python running on external computers. Ideally, most of the application HDL code and software is independent of the transport channel, so the simulated transport can be somewhat generic. Much of our transport experience is with an on-chip Ethernet/IP/UDP stack that directly attaches to a hardware PHY; a simulation of that attachment can be bridged to a workstation or CI server operating system with the tun/tap framework, allowing full (but slow) emulation of the final chip.

The transition from usual software-based testbenches to tests that involve real hardware now seems natural. Certainly developers are used to interacting with live hardware on the bench. Just like the unit-test case, we add a mandate that the developer's experience gets transformed into PASS/FAIL tests that can be refined and accumulated. On the CI side, the infrastructure now includes a toolchain for the FPGA, and tools for downloading bitfiles to the board(s). This last CI stage is now constrained to run on a physically known computer in the lab attached to the hardware made available for such testing, rather than some abstract software-only server that could be "in the cloud."

LLRF Applications

The power of CI is instrumental in our development of high-quality shared code that is used by multiple LLRF applications. Library code changes can be made with the confidence that every project using the shared sources will continue to function as designed. Fig. 2 shows a test rack at LBNL where LLRF systems for different projects coex-

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-TUAPP01

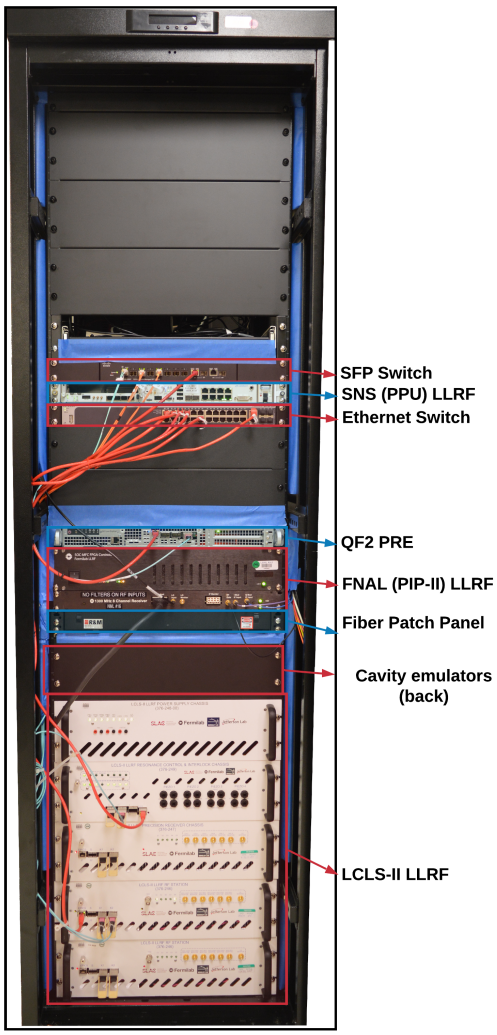

Figure 2: CI rack.

ist: LCLS-II (SLAC) [7], SNS/PPU (ORNL) and PIP-II [8] (FNAL).

The same rack also includes the networking and optical fiber equipment needed for inter-chassis and control system communications. In order to run hardware in the loop tests for LLRF applications, a hardware cavity emulator is used. The cavity emulator is essentially a crystal with similar band-pass characteristics as a resonant cavity used in accelerator (though with a wider bandwidth); it allows us to complete tests using the same cavity bring up automated scripts that are deployed in the accelerator.

SOFTWARE INFRASTRUCTURE

Dependency Management with Docker

We use several open source tools and some vendor specific closed source tools in our development workflow. Some of the open source tools are fairly stable like Python, Icarus Verilog, and Verilator. However, some are under active development like the RISC-V toolchain, LiteX, etc. Keeping track of these dependencies during development and deployment previously had been through text files and shell scripts that are tedious to maintain and potentially introduce unwanted platform dependency. As mentioned above, Docker allows us to track dependencies in a platform-independent manner. In addition to capturing dependencies, Docker is closely tied in with the CI servers, and automatically builds images with new dependencies as needed.

There are several other open source tools that perform parts of the pipeline above really well like Ansible, Chef, Salt, Puppet, Nix, etc. However only a few of them play well with existing continuous integration servers.

With Docker, one just lists all the dependencies in one file (typically named Dockerfile), and the file is usually placed in the root directory of the repository. Upon changes to this file, a typical CI server can be configured to rebuild a new image with all the dependencies. Docker is able to containerize the built image and run unit tests, integration tests, and hardware tests within this container. These containers are ephemeral: Docker creates a new one for the next round of tests. If one wants to introspect the tests, it's possible to login into the container to view the results and browse the directory structure. A typical CI system archives the results (in our case, FPGA bitstreams and executables, for retrieval and deployment). An example of the Dockerfile can be found here [9].

Another feature of Docker is that the built images can also be deployed on servers running remotely. This has it's caveats, but has potential to ease deployment in the field. Some teams have started using Docker in this manner [10].

Continuous Integration Deployment

CI servers and services tend to ping version control servers to look for changes committed on the repositories. Upon detecting changes to source code, they can be configured to do several tasks. This configuration is typically done through YAML (.yml) files. Each change in the codebase can be setup to trigger a "pipeline" (a series of tests that can be arranged in "stages"), as shown in Fig. 3.

As mentioned earlier in the paper it is up to the developer or the team to structure the tests in a meaningful sense. As it doesn't make sense to synthesize code that doesn't pass unit tests the "pipeline" can be configured to halt in case of a failure in the earlier stages.

The .yml files are expressive enough to capture rules for setting up stages, environment variables (to setup the tests), shell commands, and storing resulting artifacts (like executables and bitstreams).

SUCCESSES

So far we believe we have enjoyed successes in a variety of forms.

- All dependencies captured in one place. This was previously softly maintained through a text file. With advent of Docker, this is just as easy as having Dockerfile (a single file with instructions to install all dependencies)

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-TUAPP01

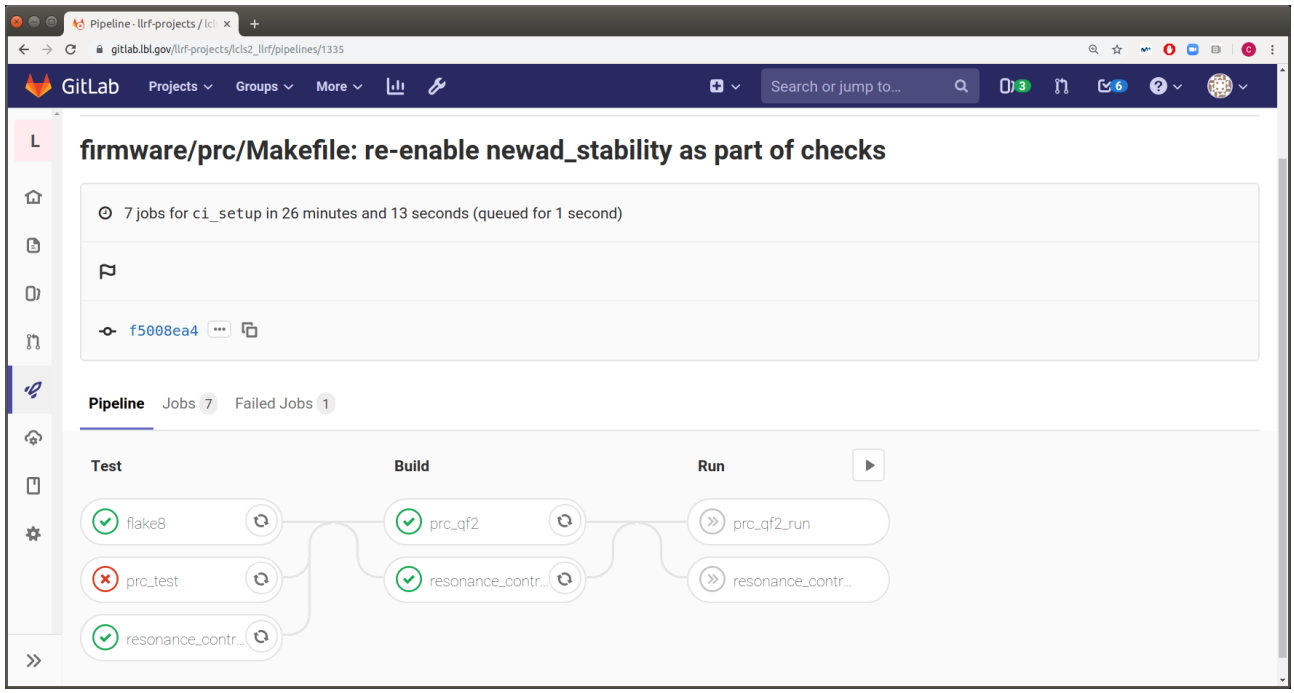

Figure 3: CI pipelines.

under version control, which then can reproducibly build the development/deployment environments.

- New features are always checked to see if they inadvertently break old features, shortening the time-to-fix, and particularly avoiding the need to spend precious machine-study time finding and fixing simple bugs. When a contributor adds a new feature to a shared codebase, the changes can break any of the unit tests, integration tests, and hardware tests. Having hardware in the loop ensures that application, infrastructure, and integration code are all tested realistically ahead of time before the code arrives in the production environment. We have found this to be a huge relief.
- Single reproducible setup accessible to all engineers. Hardware-in-loop provides at least one production-like test setup for every engineer. This is arguably superior to an engineer-specific setup which is accessible to only one engineer, when it comes to deployable code.
- Ease of keeping up with changes in dependencies. When a version of *e.g.*, iverilog or Verilator has changed, testing the entire codebase against this can be as simple as creating a new branch in version control, changing the Dockerfile to point to a later version, and committing to the codebase. This automatically, rebuilds the dependencies, and tests the code base against the changes, and keeps the image rebuilt with dependencies for future uses.

CHALLENGES

Having real hardware in the testing loop comes with its own special and exciting challenges.

- Reality of managing hardware: Our use case mandates that the machine running CI software communicates with hardware over interfaces like USB, Ethernet, PCIe, etc. When a new device gets added, removed, or unplugged by mistake, the CI software should be made aware so that the tests are run smoothly. As we expand our test setup to various projects, and platforms, we are starting to look into some creative ways to manage hardware through forms of abstractions.
- Getting locked into CI providers: So far we have chosen to self-host with open-source git hosting and CI software provided by Gitlab. However, not all features of Gitlab directly translate to other CI suppliers, including commercial hosting providers. While using any software, it is easy to get used to exclusive features, eventually losing portability to other platforms.
- Perceived difficulty in setting up the CI framework. The number of features and control knobs of a "modern" CI system are immense. They were designed to maintain and deploy large software packages with several dependencies across various platforms simultaneously. The documentation can seem large, yet lacking at the same time. That being said, we found it easy to get simple tasks up and running in a reasonable amount of time.
- Running the tests locally. CI tools provide limited mechanisms to run the same exact stages of the pipeline locally on a developer's machine. They seem to be working on it, but it's not without its quirks.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-TUAPP01

FUTURE PLANS

Considering the challenges above, we have our eyes open for better ways to produce reliable software-firmware combinations for deployment. CI and Hardware-In-Loop tests only bring us closer to that goal.

Dependency Management

We believe that dependencies can be installed in an identical way on both the local and remote machines even without containerization tools like Docker that make use of processes running in a container or a virtual machine. New package managers and operating systems are being developed that test, run and deploy code by just altering the run-time environment to point to the new dependencies. These tools also cleanly provide a way to move back and forth between the old state (in case of failure) and stay in the newly built state (in case of success). GNU Guix and Nix are such package managers.

These package managers also provide a CI mechanism which are more sophisticated, see [11].

Other features we envision covering in the future within the CI framework are automated generation of documentation and tracking metrics (such as code coverage) for testbenches.

CONCLUSION

The adoption of a CI process for the development of accelerator instrumentation has shown to have great benefits. Tests and verification of integrated designs are now automated, decreasing the cognitive load on the developers and guaranteeing a systematic verification of changes several times a day, as soon as changes are made to any part of our code base designs. Other features such as the automatic generation of artifacts (in the form of bitstreams for our FPGA builds) have proven to be useful for our controls collaborators, who are not necessarily familiar with the FPGA bitstream generation process and who would like access to those while developing software.

The CI framework, including hardware-in-the-loop tests, has been successfully deployed at LBNL. Many of the components used in that setup, along with libraries and hardware designs used in LLRF applications have been published under Open Source licenses and can be found here [12].

REFERENCES

[1] Thom Ellis, Mentor Graphics,"Increased Regression Efficiency with Jenkins Continuous Integration Before You Finish Your Morning Coffee", DVCon2017, Feb. 2017, San Jose, CA, USA.

[2] C. Serrano *et al.*, 'Open Source LLRF stack", presented at the Low-Level Radio Frequency Workshop (LLRF'19), Sep.-Oct. 2019, Chicago, IL, USA.

[3] Carlos Palminha, Synopsys, "Using 'Continuous Integration' Practices for SoC Development", https://www.synopsys.com/company/resources/newsletters/prototyping-newsletter/continuous-integration-practices.html

[4] Verilator, Open Source Verilog Simulator, https://www.veripool.org/wiki/verilator

[5] Icarus Verilog, Open Source Verilog Simulator, http://iverilog.icarus.com/

[6] RISC-V: The Free and Open RISC Instruction Set Architecture, https://riscv.org/

[7] C. Serrano *et al.*, "Design and Implementation of the LLRF System for LCLS-II", in *Proc. 16th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17)*, Barcelona, Spain, Oct. 2017, pp. 1969–1974. doi:10.18429/JACoW-ICALEPCS2017-THSH202

[8] B. Chase *et al.*, "Low Level RF Control for the PIP-II Accelerator", presented at Low-Level Radio Frequency Workshop (LLRF'17), Oct. 2017, Barcelona, Spain.

[9] Docker communication for Bedrock, LBNL's Open Source HDL Library for LLRF and DSP applications, https://github.com/BerkeleyLab/Bedrock/blob/master/Dockerfile

[10] Docker images for the Brazilian Light Source, https://hub.docker.com/u/lnlsdig

[11] Hydra: A Declarative Approach to Continuous Integration, https://nixos.org/~eelco/pubs/hydra-scp-submitted.pdf

[12] github-berkeleylab, https://github.com/berkeleylab